Decentralized Information Flow Control for Databases

by

David Andrew Schultz

MS, Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 2007

BA, Computer Science
University of California, Berkeley, 2004

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

at the

Massachusetts Institute of Technology

September 2012

Author . . . . . .
Department of Electrical Engineering and Computer Science
July 31, 2012

Certified by . . . . . .
Barbara Liskov
Institute Professor
Thesis Supervisor

Accepted by . . . . . .
Leslie Kolodziejski
Chair of the Department Graduate Committee

Decentralized Information Flow Control for Databases

by

David Andrew Schultz

Submitted to the Department of Electrical Engineering and Computer Science
on July 31, 2012, in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

Privacy and integrity concerns have been mounting in recent years as sensitive data such as medical records, social network records, and corporate and government secrets are increasingly being stored in online systems. The rate of high-profile breaches has illustrated that current techniques are inadequate for protecting sensitive information. Many of these breaches involve databases that handle information for a multitude of individuals, but databases don't provide practical tools to protect those individuals from each other, so that task is relegated to the application. This dissertation describes a system that improves security in a principled way by extending the database system and the application platform to support information flow control.

Information flow control has been gaining traction as a practical way to protect information in the contexts of programming languages and operating systems. Recent research advocates the decentralized model for information flow control (DIFC), since it provides the necessary expressiveness to protect data for many individuals with varied security concerns. However, despite the fact that most applications implicated in breaches rely on relational databases, there have been no prior comprehensive attempts to extend DIFC to a database system. This dissertation introduces IFDB, which is a database management system that supports DIFC with minimal overhead.

IFDB pioneers the Query by Label model, which provides applications with a simple way to delineate constraints on the confidentiality and integrity of the data they obtain from the database. This dissertation also defines new abstractions for managing information flows in a database and proposes new ways to address covert channels. Finally, the IFDB implementation and case studies with real applications demonstrate that database support for DIFC improves security, is easy for developers to use, and has good performance.

Thesis Supervisor: Barbara Liskov
Title: Institute Professor

Acknowledgments

Foremost, I would like to thank my advisor, Barbara Liskov, for her encouragement and support throughout my years at MIT. I learned a great deal about how to approach problems in computer systems through working with Barbara. Her guidance was instrumental in developing good abstractions and a practical model for information flow control in databases, and her careful review and many suggestions significantly improved this dissertation.

I would also like to thank my thesis committee for their help in improving this work. Sam Madden suggested developing CarTel as an example of an application that could be secured effectively with information flow control. He provided the source code, advice, and data that enabled me to produce an effective evaluation of my system in chapters 7 and 9. Nickolai Zeldovich offered numerous insightful comments that helped me make a strong case for information flow control as a basis for security.

I am grateful to the members of the Programming Methodology Group at MIT for their support as well. I would especially like to thank Dan Ports for his encouragement, and for many helpful comments and discussions about this work. Ben Vandiver convinced me early in my dissertation research that database security is an interesting problem to work on, and he provided me with his Java-based TPC-C implementation, which I used to benchmark an early prototype of my system.

Eddie Kohler improved this thesis in two ways. First, as the author of HotCRP, an application I use as an example in chapter 7, he validated a security flaw I found and provided pointers on past HotCRP information leaks to study. Second, he unknowingly improved the visual appearance of this document by writing the free LCDF Typetools software package, which supplies the necessary wizardry to shoehorn modern font technology into the LaTeX typesetting system.

Andrew Myers and his research group at Cornell made the lab a livelier place during their visit to MIT, and they also provided useful feedback and advice regarding this work. Andrew shared his keen insights and perspective on decentralized information flow control. He also taught me how to use noninterference to think about covert channels, which helped me develop safe semantics for foreign key constraints.

Additionally, I would like to thank Mike Stonebraker and the many others whose feedback has strengthened this work. I am also indebted to the folks at the MIT library for their valiant efforts to track down articles about some of the early work

on information flow control in databases, many of which are out of print.

On a more personal level, I am deeply indebted to God and family for all of their love and support throughout my life. My parents, Jean and Randy, and my sister Laura, have always been there for me in good times and bad. Stacy Fahrenthold helped keep me sane and well-nourished during the final months of my doctoral work by cooking delicious dinners for me when I had no time. Before those final months, my friends and mentors in the MIT Tae Kwon Do Club pulled me away from my computer and kept me in good physical shape. Furthermore, I could not have done it without all my other friends at and around MIT.

Contents

List of Figures

Chapter 1 Introduction

Online applications are becoming increasingly sophisticated, sharing and processing data for many individuals in increasingly sophisticated ways. As these systems become more complex, so too do the security problems. Medical systems must provide clinicians, receptionists, accountants, medical researchers, auditors, insurance processors, and lab technicians with the data they need to get their jobs done, but the systems must also protect confidential patient data from misuse [1]. Social networking websites are under increasing pressure from users to provide better tools to protect privacy, but the implementation of such constraints has proved to be error-prone [30]. Each application's data processing needs are unique, so responsibility for ensuring data security typically falls on the application. Since both the applications and the security policies are complex, information leaks and other security holes abound.

Despite the fact that there has been much work on improving the security of these applications, for instance, by eliminating potential SQL injection attacks [64, 98, 118, 135], data breaches are still common [23, 48, 147]. For example, 470,000 individuals' medical records were exposed by a recent breach of a large health insurer [111]. The breach resulted from a vendor adding a new component to the insurer's website, but inadvertently omitting the check to ensure that customers could view only their own records. Unlike SQL injection vulnerabilities, which can be detected automatically or avoided with abstractions such as prepared statements, the missing check is an error of intent: the buggy insurance application was semantically

valid, but did not enforce the intended security policy. Such logic errors are inevitable as long as people continue to rely upon entire applications to enforce security policies. This dissertation shows that it is easy to address these types of leaks by shifting responsibility for controlling the use and release of information into the platform, trusting only a minimal piece of the application.

Enforcing a security policy in the platform restricts what applications can do, so for acceptance, it is critical to choose an abstraction that is not overly burdensome. Systems typically use access control to protect confidential data, and in particular, modern databases management systems have mechanisms to enforce sophisticated access control policies [120]. In access control systems, principals are only allowed to access data for which they are authorized. However, developers often reject database-enforced access control in favor of less secure alternatives because the kinds of restrictions that would be needed to protect user data from application bugs are too inconvenient. In particular, if the database doesn't trust the application with sensitive data, many data processing tasks that would be more convenient to perform in the application must take place in the database instead.

Information flow control (IFC) provides an abstraction that can express many security policies more conveniently and securely than access control. Rather than restricting access, the platform tracks flows of sensitive data and prevents data from being used or released inappropriately. IFC is convenient because it allows sensitive data to be processed in both the application and the database. It improves security because it can enforce end-to-end security policies. For example, in a medical information system, IFC can enforce the policy that the medical records for a patient, Alice, are never visible to any other patient. The information flow policy need not say anything about intermediate processing tasks, such as the one that checks Alice's medication list for possible drug interactions. In contrast, an access control policy couldn't prevent bugs in the drug interaction checker from compromising Alice's security, except by carefully reasoning about and sandboxing that module.

This dissertation presents a new approach to database security based on IFC. The remainder of the introduction shows why approaches to database security based on access control are inadequate, presents the new approach, and explains the principal contributions and results.

1.1 The Case for Information Flow Control

Information flow control is a good way to enforce many security policies that cannot be expressed easily with access control. Access control specifies what resources can be used, whereas information flow control makes it possible to control *how* they may be used. The following examples of security goals in a medical clinic have a natural interpretation as information flow policies, but would be tedious to achieve with access control:

— *Appointment notifications.* A secretary may, in principle, require access to at least part of every patient's record at a clinic, and he also requires the ability to send email about appointments. However, he should not be able to accidentally or willfully disclose medical information via email, or to parties not involved in each respective patient's care.

— *Exam room scheduling.* The facilities supervisor for the clinic may need to know when an examination room is unused so that repairs can be made. Producing this information requires read access to patients' appointment schedules, but the software should not release information about the individual appointments to the supervisor.

— *Third-party data analysis.* The clinic hires a data analysis firm to assess the effectiveness of the clinic's cost controls. The clinic has a legal obligation to ensure that the firm does not misuse confidential patient data [1], whereas the firm does not want to release its proprietary analysis tools to the clinic. Simultaneous access to both the data and the tools is required, but only the aggregated analysis results should be released.

Since each example involves data that must be accessed but not released, some form of sandboxing is needed. Sandboxing the application in an access control system is difficult: after all, the application must still be able to interact with the database. Leaks could still occur if data were written to the wrong part of the database, as happened in a recent incident involving a large hotel chain [115]. Alternatively, the developer could move the code that processes sensitive data into stored procedures in the database. In some applications, moving more computation into the database may be appropriate; in others, however, it is inconvenient, and forcing developers to build the system in unnatural ways works against the goal of security.

Moreover, moving sensitive processing tasks into stored procedures doesn't solve the problem. Even though those procedures might be restricted from communicating with the outside world, they can still write to the database. In the third-party data analysis example, for instance, the analysis firm's software might write patient data into a table containing public information – where it might later be exposed. Mitigating that kind of bug with access control requires careful reasoning about the read and write privileges that must be granted to each code module. This approach is time-consuming to implement, and hence is not commonly used in practice. Even if it were, an access control policy that tightly constrains the privileges of each stored procedure in a large application would be complicated; therefore, the policy itself could easily be incorrect.

1.2 Information Flow Control

Information flow control (IFC) has been gaining acceptance as a better methodology for protecting privacy without unduly restricting access to sensitive data. IFC addresses the type of problem illustrated in the aforementioned examples by annotating data records with labels describing their sensitivity. Rather than restricting access, information flow control systems instead track data as they propagate, and protect privacy by preventing sensitive data from being released from the system improperly. In addition to privacy, IFC can also protect integrity; it does so by ensuring that trusted data cannot be influenced by untrustworthy (that is, low-integrity) sources. Integrity is discussed in more detail in chapter 3.

IFC was introduced in the mid-1970's [9, 11, 32], but it has not achieved wide-scale adoption in the private sector. Its lack of popularity is due in part to the fact that the original systems were based on multi-level security (MLS) and mandatory access control, as advocated by the Orange Book [39]. MLS systems use broad labels such as *confidential*, *secret*, and *top secret*. Furthermore, they enforce centrally administered security policies; consequently, while these systems may be suitable for organizations with well-defined hierarchies such as militaries, they do not adapt well to settings where users have diverse security interests.

Most of the recent work in IFC has advocated decentralized information flow control (DIFC), introduced by Myers and Liskov [113]. In DIFC, data classification is more specific than in the MLS model: the system distinguishes the security concerns of Alice's medical records from those of Bob's, for example. Furthermore, policies are

discretionary instead of mandatory, meaning that individual principals can specify the security policies for their own data.

Prior DIFC research focuses on providing a labeling and flow control policy framework either at the level of the operating system [85, 86, 152] or at the level of the programming language environment [21, 94, 112, 113, 125, 150]. However, much of the data flow within currently deployed systems takes place within a database management system (DBMS), and existing DIFC approaches do not adequately capture what happens within the database. These systems either do not address persistence directly, or they extend the file system to support information flow; however, the relational model provides a better abstraction for many applications [24].

This thesis fills a gap by bringing the DIFC model to database systems. It extends the work on multi-level-secure databases by providing semantics appropriate for DIFC, and it complements the work on DIFC operating systems and programming languages by providing a persistent, DIFC-aware relational store.

1.3 IFDB: Secure Data Processing with DIFC

The IFDB system introduced in this dissertation is the first to bring end-to-end information flow control to DBMS-backed applications. Information from the outside world enters the platform through applications, which process it and store it in the database. Subsequently, the data may be processed by other applications, and by computations within the DBMS, such as views, stored procedures, and triggers. The platform tracks these flows and enforces a uniform security policy throughout the data's life cycle. To capture the entire computation history of the data, IFDB works with application runtime environments that also support information flow control. Two such environments, PHP-IF and Python-IF, were built to interact with IFDB.

IFDB is also the first DBMS to use a DIFC model. Support for DIFC is important for many applications, particularly web services, because DIFC allows users to control how their information is used. The IFDB architecture is not tied to a particular programming language or a particular DIFC model or implementation. However, in the interests of concreteness, this dissertation describes an IFDB prototype based on a particular DIFC model, namely the Aeolus model [17, 18]. The Aeolus model is described in chapter 3.

As noted at the beginning of this chapter, developers tend not to use security mechanisms if they are too burdensome. IFDB's interfaces are designed to be easy to

program. The system works with existing languages, such as SQL, PHP, and Python, with straightforward extensions to support DIFC. Adoption of the Aeolus model also enhances usability; security policies are expressed in terms of delegation and exercise of authority – concepts programmers are familiar with.

Of course, performance is also crucial to the acceptance of the system. IFDB minimizes overhead by tracking information flows on a coarse, per-process granularity within the application platform (where fine-grained tracking would be expensive) and fine-grained, per-tuple tracking only within the database. The key observation in support of this design choice is that the database is the primary shared medium through which leaks could occur. Therefore, fine-grained tracking within the DBMS is essential for security.

Integrating the DIFC model into a relational DBMS presents several new challenges. First, declarative queries require a different kind of reasoning than earlier DIFC work, which generally relies on file systems as the persistent store: for example, without appropriate precautions, a query for records about hospital patients who don't have cancer can implicitly reveal which patients do have cancer. A second challenge is that adding DIFC should not sacrifice data independence, for instance, by forcing developers to decompose tables based on the sensitivity of the information they contain. Third, without special consideration, important DBMS features such as transactions and constraints can lead to information leaks via covert channels. IFDB addresses these challenges as follows:

— It introduces the *Query by Label* model (chapter 4), which provides a practical way to do relational queries while respecting information flow rules.

— It includes new abstractions, *declassifying views* and *endorsing views* (section 4.5), which help to retain data independence. In particular, they ensure that database designers can refer to and partition their data in a logical way, even if that means combining information with different security requirements.

— To handle potential covert channels in transactions, IFDB introduces new semantics based on two ideas: *transaction commit labels* and *transaction clearance* (chapter 6). For constraints, IFDB adds *DECLASSIFYING and ENDORSING clauses*, and also adopts *polyinstantiation* (chapter 5).

1.4 Organization

This thesis introduces the IFDB architecture and model over the course of five chapters. Chapter 2 presents a brief, high-level overview of the components of the system, and explains how they interact with one another. It also defines the threat model, and outlines the security goals for the system. Chapter 3 reviews IFDB's information flow model, which is based on an earlier DIFC system called Aeolus [17, 18]. Chapters 4 to 6 cover the IFDB interface, which is the central contribution of this dissertation. Chapter 4 explains IFDB's basic Query by Label model, which extends the relational model with support for decentralized information flow control. Chapters 5 and 6 describes how IFDB addresses some of the problems with supporting constraints and transactions, respectively, in an information flow system. (Some additional topics on transactions and information flow are covered in appendix A.)

Chapter 7 describes two applications, HotCRP and CarTel, which have been modified to use IFDB. It recounts the author's experiences using IFDB with these applications, and explains how IFDB prevented real security vulnerabilities in these applications from leaking information.

Chapter 8 details the implementation of IFDB, which uses a modified version of PostgreSQL 8.4.10. It also addresses security concerns regarding the implementation, such as reducing the size of the trusted computing base and mitigating timing channels. Chapter 9 shows that the implementation performs well for the applications studied. Microbenchmarks are presented to provide further intuition into the performance results.

IFDB draws on ideas about information flow control developed over the last four decades. Chapter 10 reviews the work that influenced IFDB, as well as research that addresses complementary problems. Finally, chapter 11 concludes this dissertation with a summary of the major contributions and some ideas for future work on DIFC in database systems.

Chapter 2 System Architecture

IFDB is a database management system (DBMS) with a new security architecture based on DIFC. The security architecture ensures that queries, stored procedures, and other computations running inside the DBMS respect a specified information flow policy. However, most computations extend beyond the DBMS; typically, an application issues multiple queries and performs its own computations on the output, often producing a result for the user. Therefore, IFDB is designed to integrate with application runtime environments that also support DIFC. IFDB and the application runtime work together to ensure that the entire computation respects a common information flow policy.

This chapter shows what a complete IFDB deployment looks like, and explains how IFDB interacts with database clients. Since IFDB presumes that the clients support DIFC, the trusted computing base (TCB) includes parts of the client platforms as well. Section 2.3 describes the security assumptions and their implications, including IFDB's approach to covert channels.

2.1 The Information Flow Platform

An IFDB deployment consists of a single DBMS and potentially many database clients. The clients themselves are applications, which typically communicate with external users over the network. Commonly, the database clients provide a web service, and external users interact with it via web browsers.

The DBMS and clients are part of a single information flow platform, and they all support a common DIFC model. Within the system, the platform tracks the secrecy and integrity of every data object and every process. The platform enforces a security policy, which restricts how data may be used: secret data should not be released inappropriately, and low-integrity data should not be trusted. In particular, the DIFC platform confines applications and controls how they can communicate with external users, who are outside the system and therefore not subject to the information flow constraints.

The platform also mediates all communication with the DBMS, as well as all communication among applications running within the system. The IFDB DBMS allows connections only from clients operating under the platform. Platform nodes communicate via extended protocols that transmit label information to allow the system to track flows.

Figure 2-1 illustrates an IFDB deployment with several clients and external users. The DIFC runtime sandboxes applications and interposes on all communication. The system tracks the secrecy and integrity of data flowing among applications, and between applications and the database. It also tracks flows that occur inside the database, for instance, due to the actions of stored procedures, which are considered to be application code. The DBMS and the application servers also communicate with an authority service. The following section explains the purpose of the authority service, and introduces two DIFC runtime environments that support IFDB.

2.2 Platform Support for the Aeolus Model

The IFDB architecture is not tied to a particular programming language or a particular DIFC model or implementation. However, in the interests of concreteness, the IFDB prototype is described in the context of a particular DIFC model, namely the Aeolus model [17, 18]. This section describes the architectural components that were introduced to support this model, while chapter 3 describes the model itself.

All of the platform nodes, including IFDB, rely on an authority service. Information flow policies in Aeolus are based on use of authority. The present IFDB prototype integrates the authority service into the database. This approach improves performance because the database has direct access to the information, and the database clients don't need to make separate connections to the authority service and the DBMS.

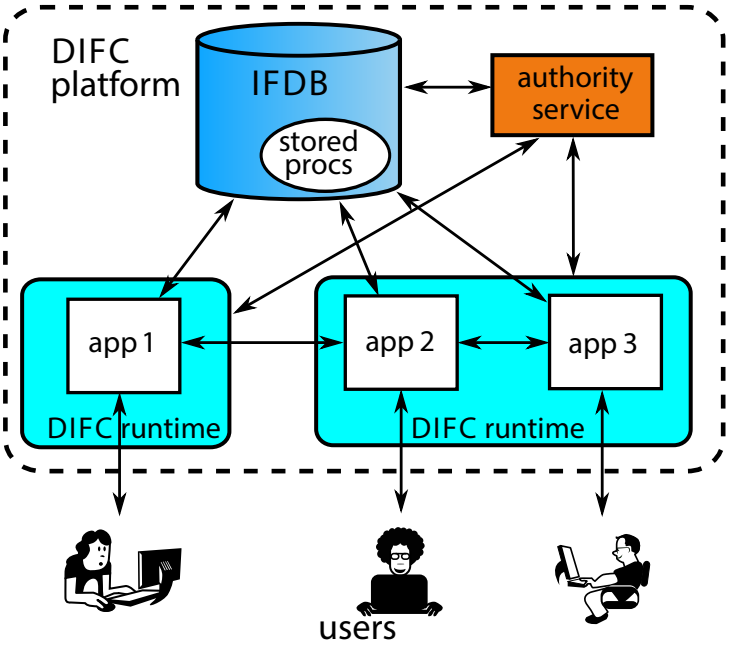


Figure 2-1: In this IFDB deployment with three applications running on two application servers, the dotted line represents the perimeter of the information flow platform. The shaded components constitute the trusted computing base for the platform.

The platform includes two runtime environments that support IFDB and the Aeolus model: PHP-IF, an extension of PHP, and Python-IF, an extension of Python. A Java implementation of Aeolus is also available, but has not yet been integrated with IFDB. The runtime environments in figure 2-1 could be PHP-IF or Python-IF. PHP-IF and Python-IF constrain how the applications they run can communicate, as described in the preceding section. They also support the Aeolus API and an interface to the DBMS via the IFDB database driver.

Programs – specifically stored procedures – also run within the DBMS. Stored procedures are often used to perform complex data processing tasks on behalf of applications, so it is important that the security policy apply to stored procedures as well. Therefore, IFDB implements another version of the Aeolus API as an extension to SQL. Thus, application code running in the database is able to manage information flows as well.

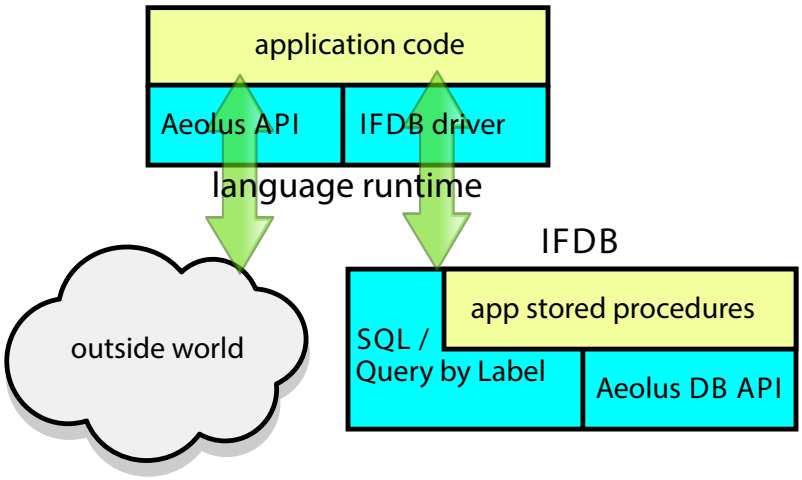


Figure 2-2: Applications interact with the outside world via the Aeolus API, and with the database using IFDB's Query By Label model.

Figure 2-2 illustrates how applications interact with the platform. All communication occurs through the Aeolus API and the database driver. Applications issue queries in SQL, with the Aeolus extensions. The Query by Label model enforces information flow restrictions on these queries, as described in chapter 4. Applications also invoke stored procedures written in a procedural extension to SQL, and these procedures can also use the Aeolus API to make information flow decisions.

2.3 Platform Security Assumptions

IFDB is intended to prevent application bugs from compromising security, but to do that, the platform itself must be secure. This dissertation defines the trusted computing base (TCB) for the system as the set of components that must function correctly to ensure that the specified security policy is enforced. The TCB includes the information flow platform itself and all the layers below it. Specifically, the DBMS (IFDB) and language runtimes (PHP-IF and Python-IF), as well as the operating systems and hardware on which they run, must be trusted. Much prior research, some of which is covered in chapter 10, has addressed the problem of producing secure database systems, language runtimes, and operating systems; IFDB is complementary to this work, and reducing the trusted base further is a promising topic for future research.

The system design also presumes secure channels [90] among platform nodes. An attacker should not be able to learn the contents of communications between nodes, modify messages, or impersonate platform nodes. One way to implement secure channels is to colocate the database and application servers and connect them via a trusted switch. However, for wide-area deployments, a cryptographic implementation of secure channels based on TLS is supported as well.

The users and applications are not trusted by the platform; the platform constrains applications to follow the security policy, subject to the caveats about covert channels described in the following section. Of course, users' data are secure only insofar as the policy is correctly specified. Section 3.4 explains how to reason about security policies and application correctness in the IFDB model.

2.3.1 Covert Channels

Although the platform prevents applications from violating the information flow rules directly, IFC systems are vulnerable to policy violations via *covert channels*, which are vectors for unintended information leaks [89]. For instance, a process might leak partial information about Alice's password by exhausting a shared resource if and only if the first bit of the password is a 1. A collaborating process can then learn the first bit of Alice's password by observing that the resource is unavailable. However, since this is not a normal communication channel, the system does not track the information flow.

This dissertation's approach to handling covert channels is inspired by the field of side-channel cryptanalysis, in which a distinction is drawn between covert channels in the abstract model and *side channels*, which are attacks on the implementation [124]. The model exhibited in this dissertation is intended to be free of covert channels. In the IFDB implementation, however, the amount of time operations take to complete is affected by access to shared resources, so inevitably the implementation will be vulnerable to attacks. Side-channel attacks on database implementations are a serious problem even without IFC [51], but nevertheless, it is important to confine these attacks to the implementation. Doing so makes it possible to apply mitigation techniques, such as quantizing response times, without changing the semantics of the system. Section 8.5 discusses attacks against the implementation, and how they can be mitigated.

Chapter 3 Information Flow Model

IFDB's DIFC model is based on the abstractions introduced by Aeolus [17, 18], a platform for secure distributed computation. This chapter reviews the Aeolus information flow model. Chapter 10 describes alternatives to the Aeolus model. Many of the contributions of this dissertation, including Query by Label, declassifying views, and transaction commit labels, are applicable to other DIFC models as well.

DIFC systems classify data based on two types of concerns: *secrecy* and *integrity*. Secrecy has to do with release: To whom, and under what circumstances, should the data be sent out? An example of a secrecy requirement is that Bob's medical records should not be released to anyone except for Bob and the doctors involved in his care. Integrity is a statement about trustworthiness: For what purposes should the data be trusted? For instance, the notion that Bob's prescriptions should only be filled if they were written by a doctor is an integrity requirement.

The Aeolus model is an evolution of the decentralized label model introduced by Myers and Liskov [113]. The model is based on several key concepts: labels, which identify the classification of each data object; principals, which represent entities with security interests, such as users and roles; and an authority structure, which is instrumental in expressing the information flow policy. The policy is embedded in the application: decisions about what information should be trusted or released are made by code running with authority. Aeolus provides abstractions that are intended to reduce the amount of trust that must be placed in code. The following pages explain the model, and figure 3-1 summarizes the essential parts of the interface.

3.1 Tags and Labels

Aeolus uses *tags* to represent classes of sensitive information. A tag is associated with all of the data that has similar secrecy or integrity restrictions. For example, all of Bob's medical records might carry the bob-medical tag, and all of Bob's financial records might carry the bob-financial tag. Alice's records might be similarly labeled with tags corresponding to Alice. In this way, Alice's information can be protected from Bob, and vice versa.

Some computations, such as one that computes statistics over patient records in a medical clinic, are run over data with many different tags. Aeolus supports *compound tags*, which provide convenience and representational efficiency for such computations by statically grouping tags and treating them as a unit. For example, the bob-medical tag is a member of the all-patients-medical compound tag.

Each data object and each process has a *secrecy label* and an *integrity label* to identify their respective secrecy and integrity requirements. Each label is a set of zero or more tags. Aeolus tracks labels as data propagates through the system, and helps to ensure that derived data are properly labeled. For instance, if Bob's medical bill is derived from sources that contain both his financial and medical information, it should have a secrecy label that includes both the bob-medical and bob-financial tags. The following section explains how Aeolus enforces this property.

3.2 Processing in Aeolus

Each process and each data object in the system has a secrecy label L_S and an integrity label L_I. Object labels are immutable; they are specified when the object is created and cannot be changed later. Process labels, in contrast, change over time to reflect the secrecy and integrity of the information the process has read.

A single rule governs reads and writes of objects, as well as communication between processes. Data can flow from source A to destination B if the secrecy tags of A are a subset of the secrecy tags of B, and the integrity tags of B are a subset of the integrity tags of A:

Rule 3.1. (Safe Information Flows)
Data can flow from A to B iff $A.L_S \subseteq B.L_S$ and $A.L_I \supseteq B.L_I$.

Rule 3.1 ensures that the label of each object reflects the tags of all the data that produced it, and the label of each process reflects the tags of all the data that the

process read. In particular, the secrecy label of the process must expand to include the secrecy tags of all the objects the process reads, and the integrity label must shrink to reflect a lower bound on the trustworthiness of the data the process reads. Processes are restricted from writing to destinations with lower secrecy or higher integrity than themselves. In effect, processes are contaminated by what they read, and that contamination restricts how they can communicate. This notion of restriction leads to a definition of a partial ordering for a secrecy/integrity label pair:

Definition 3.1. (Label Ordering)
A pair of labels $(A.L_S, A.L_I)$ is *less restrictive than* another pair $(B.L_S, B.L_I)$, written $(A.L_S, A.L_I) \prec (B.L_S, B.L_I)$, iff $A.L_S \subset B.L_S$ and $A.L_I \supset B.L_I$

The less-restrictive-than relation[1] captures both secrecy and integrity in a single statement, and is used herein to simplify the presentation. Information can safely flow from A to B provided that A's label pair is no more restrictive than B's label pair, written $(A.L_S, A.L_I) \preceq (B.L_S, B.L_I)$.

The flow rules also restrict the release of information to the outside world, which of course is not governed by IFC. In Aeolus, external devices such as remote machines, displays, keyboards, and printers are regarded as having empty secrecy and integrity labels. Therefore, a process may not release information to these devices if it is contaminated by any secrets, and if it reads from an external source, it will have no integrity.

3.3 Label Changes and Authority

Raising the label pair of a process, making it more restrictive, is always safe from an information flow perspective. (The notion of *clearance*, which limits how far a process can raise its label, can be useful. Clearance is discussed in section 4.6.) With a higher label pair, the process may be able to read more sensitive information, but it will be unable to release it. However, since an empty label is required to release information, there needs to be a way to lower a process label as well; otherwise there would be no way to get labeled data out of the system!

1. The literature on multi-level security often uses the term *dominance* to describe the same concept: more restrictive label pairs strictly dominate less restrictive ones. This dissertation uses *restrictiveness*, because it conveys the fact that processes are contaminated by what they read, and their contamination limits what they can do.

Principals

createPrincipal() → P. Return a new principal. The caller's principal acts for the new principal P.

actsFor(P_1, P_2). Add an acts-for link from P_2 to P_1. The caller must act for P_1, and the link must not create a cycle in the acts-for graph.

revokeActsFor(P_1, P_2). Remove the acts-for link from P_2 to P_1, if one exists. The caller must act for P_1.

Tags

makeTag() → t. Return a new tag. The caller's principal is authoritative for the new tag t.

makeSubtag(t_1) → t_2. Return a new tag t_2, which is a subtag of compound tag t_1.

grant(t, P_1, P_2). Add a delegation link for t from P_1 to P_2. P_1 must be authoritative for t, the caller must act for P_1, and the link must not create a cycle in the delegation graph.

revokeGrant(t, P_1, P_2). Remove the grant for t from P_1 to P_2 (if one exists). The caller must act for P_1.

Labels

addSecrecy(t). Add tag t to the secrecy label of the process.

declassify(t). Remove tag t from the secrecy label of the process. If t is a compound tag, any subtags of t are also removed. The caller must be authoritative for t.

removeIntegrity(t). Remove tag t from the integrity label of the process. If t is a compound tag, any subtags of t are also removed.

endorse(t). Add tag t to the integrity label of the process. The caller must be authoritative for t.

Figure 3-1: A summary of Aeolus operations

Authority Closures and Reduced Authority Calls

makeClosure(λ, P) $\rightarrow C$. Return a closure that, when invoked, runs procedure λ with principal P's authority. The caller must act for P.

C(args…) $\rightarrow r$. Invoke closure C. The closure runs with the authority of P, the principal bound to the closure, but inherits the caller's labels. On return, the process labels are merged with the caller's original labels: the secrecy labels are unioned, and the integrity labels are intersected. This ensures that closures can add contamination (for instance, by reading and returning a secret), but they cannot inadvertently remove the caller's contamination.

call(P, λ, args…). Call the procedure λ with the given arguments, running with the (reduced) authority of principal P. The caller must act for P. The caller's authority is restored when P returns.

Figure 3-1: A summary of Aeolus operations (continued)

Therefore, Aeolus also provides operations to *lower* label pairs by removing secrecy tags and adding integrity tags. The former operation is called *declassification* and the latter *endorsement*. Consider a statistics package that aggregates medical records and outputs scrubbed data, which presumably does not reveal a significant amount of personally identifiable information. It is therefore appropriate for the scrubbed data to have a smaller secrecy label than the source data does. Thus, after the process runs the statistics package, it should declassify to remove tags from its secrecy label before writing the aggregate result.

Declassification and endorsement make information leaks possible. Since they are not safe in general, they are privileged operations. Authority in DIFC is discretionary; privilege is vested in *principals*, which represent users and roles. A principal must have authority for a tag in order to declassify or endorse it.

Each tag has an owning principal, which has authority for the tag. A principal that has authority for a tag can grant that authority to other principals, and later revoke that delegation. For example, Bob could delegate authority for the bob-medical tag to his doctor. A principal P can also allow another principal Q to *act for* it; this allows Q to do anything P can do. Acts-for links can be revoked as well. Acts-for relationships provide a means to implement groups and roles [90].

Process label changes are explicit in Aeolus. A process must adjust its label using the addSecrecy() and removeIntegrity() operations (see figure 3-1) prior to reading a sensitive object. Similarly, in order to make its label less restrictive, a process must exercise its authority and call declassify() or endorse(). This approach contrasts with some earlier information flow systems, such as ADEPT-50 [146], IX [106], and Asbestos [85], in which process labels adjust dynamically to reflect the contamination of data they have read. Making safe label changes explicit is important because it prevents an important covert channel. In systems that allow automatic label changes, the label change itself can convey information about a secret [34, 84]. Requiring *unsafe* label changes to be explicit is also important, because if declassification and endorsement were implicit, programs might exercise their authority and enable information leaks accidentally.

3.4 Defining the Policy

An information flow policy is shaped by controlling the circumstances under which declassification and endorsement occur. The Aeolus model provides a framework for enforcing the policy, and for reasoning about security. For instance, if Bob's medical records are properly tagged when they are input initially, then the only parts of the program where they could be leaked are the places where the bob-medical tag (or its supertag, the all-patients-medical tag) is declassified. Other code does not even need authority to declassify the tag, even if it is involved in processing Bob's records.

However, Aeolus merely provides a mechanism and does not dictate what the policy itself should be. Referring back to the statistics package, an obvious question is: How do we know that the aggregate results do not reveal a significant amount of personally identifiable information? This type of question has been well studied, and section 10.7 surveys some of the work in the area.

3.5 Support for the Principle of Least Privilege

The Principle of Least Privilege [126] is essential for building secure applications, because it prevents bugs from becoming security failures. It dictates that the application should operate using as little privilege as possible. The delegation and acts-for relationships presented in the preceding section help support the Principal of Least Privilege by providing a fine-grained way to control the authority vested in each

principal. However, it is also important to provide abstractions that minimize the privileges available to each part of the application.

Aeolus provides two mechanisms to support the latter goal. *Reduced authority calls* allow an application to invoke a procedure that will run with the authority of a less-privileged principal than the caller. The caller can specify any principal that it acts for. *Authority closures* are procedures that are bound to a specific principal that the creator acts for when they are created. When they are invoked, they run with the authority of that principal. They provide a means to associate authority with a specific privileged action. For example, a CheckPassword closure might read the password file to authenticate a user, declassify, and release only the outcome (authentication success or failure). Thus, no other code in the program needs to have authority to declassify (and potentially leak) information in the password file.

Authority closures can raise the label of the process and then use authority to lower it again, but they cannot remove contamination from their callers. When an authority closure call returns, the process labels are merged with the labels the process had at the start of the call: the secrecy labels are unioned and the integrity labels are intersected. This merging ensures that the closure cannot be abused to remove contamination that the caller had previously.

3.6 Boxes, Shared Volatile State, and Files

The Aeolus platform supports several additional abstractions for sharing persistent and volatile state. Boxes provide a way to encapsulate sensitive information so that it can be passed through intermediaries without contaminating them. Shared volatile state provides an efficient way for processes on the same machine to share information in a way that respects the information flow rules. Aeolus also supports labels on files and directories, with the appropriate semantics as implied by rule 3.1.

Boxes, shared volatile state, and files are not relevant to the IFDB model, but the language runtimes that interact with IFDB, such as PHP-IF and Python-IF, could support them. In particular, web services increasingly rely on soft state and caching mechanisms such as memcached [47] to increase scalability, and Aeolus' shared volatile state abstractions may prove useful in that context. However, the web applications studied as examples in chapter 7 use databases, not file systems, as their backing stores. They do not require inter-process communication or sophisticated caching mechanisms.

Chapter 4 Query by Label

Reasoning about flows of sensitive information can be challenging, and the *ad hoc* techniques in use today, often based on access control, are error-prone. For example, it has been alleged that Google violated its own privacy policy when it introduced its Buzz social networking service. When users of Google's GMail service signed up for Buzz, Google used information from their private chats, emails, and contact lists to generate "follower" lists, which were published on their public profiles. According to an FTC complaint, these lists often contained "individuals against whom they had obtained restraining orders, abusive ex-husbands, clients of mental health professionals, [...] and recruiters they had emailed regarding job leads." [45] The mistake isn't surprising: the disclosure involved multiple services developed by different engineering teams, and the Buzz developers likely weren't aware of the privacy policy governing the GMail data.

IFDB uses a new query model called Query by Label, which is intended to prevent similar mistakes in database-backed applications. The core design principle in Query by Label is that *all information releases must be explicit*. Specifically, any unsafe information flows must be accompanied by an explicit use of authority (declassification or endorsement) to assert that the disclosure is acceptable.

Query by Label integrates the Aeolus DIFC model described in chapter 3 with the relational model. Client applications running under an Aeolus-based language runtime connect to the DBMS, and the DBMS collaborates with the language runtime to ensure that the application follows the information flow rules. As described in

the preceding chapter, each process has a secrecy/integrity label pair, which reflects the contamination of all the sensitive data the process has read thus far. The process labels are a key part of the Query by Label model; they affect what the process can read from the database, and what it is allowed to write.

This chapter begins in section 4.1 by reviewing the tenets of the relational model. Sections 4.2 and 4.3 develop the basic Query by Label model and explains how it ensures that all unsafe flows must be vouched for via declassification or endorsement. Sections 4.4 and 4.5 introduce new abstractions, stored authority closures and declassifying views, that make it convenient to vouch for flows. Finally, section 4.6 explains the relationship between the DIFC-based approach presented in this chapter and access control.

4.1 The Relational Model

The relational model [24, 25] was introduced by Codd in 1970, and is the dominant logical data model today. It provides a scheme for representing data and a set of operators to access the data. This section briefly reviews the model to introduce terminology and syntax, and to discuss some of the properties the model provides, such as data independence and consistency. Supporting these properties in an information flow system is more challenging; section 4.5 and chapter 5 explain the issues and how Query by Label addresses them.

4.1.1 Data Representation

Data in the relational model are represented as sets of n-tuples, as follows:

Attributes (also known as *fields* or *columns*) are atomic units of data over some domain. For example, bob is an attribute over the domain of names, while 8/23/1923 is an attribute over the domain of dates. Domains may include a special *null* value, which is used to indicate information that is missing, typically because it is unknown or inapplicable.

Tuples, or *rows*, are ordered collections of attributes. Tuples typically represent relationships between attributes. For instance, (bob, 8/23/1923) is a tuple that relates Bob's name to his date of birth.

Relations are unordered sets of tuples. Each relation has a schema, which defines a name and domain for each attribute in the relation; all tuples follow the schema. For example, the schema for a relation containing names and dates of birth (DOBs) could be (name : string, dob : date).

Since relations are intended to be analogous to mathematical sets, they do not allow duplicate tuples. Additionally, each relation has a *primary key*, which is a collection of attributes that uniquely identifies any tuple in the relation. The analogous concept of *tables* in most database systems optionally allows for duplicates, in which case there may be no primary key. (This is called *bag semantics*, as opposed to set semantics.)

4.1.2 Relational Operators

The model defines a relational algebra for retrieving data. The algebra consists of a set of primitive operators, which can be composed to produce complex queries. The following operators are the most commonly used:

$\textsc{Select}_P(R)$ filters tuples in relation R, producing a new relation that includes only the tuples matching predicate P. The predicate is boolean expression evaluated on the attributes of the tuples – for example, age > 18.

$\textsc{Project}_{A_{i_1},\ldots,A_{i_n}}(R)$ extracts attributes A_{i_1} through A_{i_n} from all tuples in relation R. In the formal model, duplicates are discarded; however, the analogous operation in ANSI SQL does not do so by default.

$\textsc{Join}_P(R, S)$ produces a relation consisting of the concatenation of every pair of tuples in R and S that satisfies predicate P. Typically the predicate specifies that some attributes in R should be equal to some attributes in S – an *equi-join*. For example, suppose that relation R consists of (patient_id, name, dob) tuples and relation S consists of (patient_id, address) tuples. Then the expression $\textsc{Join}_{R.\text{patient\_id}=S.\text{patient\_id}}(R, S)$ evaluates to a set of (patient_id, name, dob, address) tuples. (The patient_ids in R and S must match, so only one copy of this field is typically retained.)

In addition to the join operator described above, which is often referred to as an inner join, there are three varieties of *outer joins*. $\textsc{LeftOuterJoin}_P(R, S)$ produces the same tuples as $\textsc{Join}_P(R, S)$ plus any additional tuples in R that

have no matching tuples in S. For these tuples, the missing S-attributes are null. $\textsc{RightOuterJoin}_p(R, S)$ is defined analogously, with the roles of R and S reversed. $\textsc{FullOuterJoin}_p(R, S)$ produces the union of all the tuples in the left and right outer joins.

4.1.3 Constraints

Integrity constraints[1] ensure that data in the database are consistent. Constraints protect against application bugs that might corrupt the database, and they ensure that applications see results of the expected form. The taxonomy and terminology presented here differ from Codd's original formulation.

Domain constraints, or *row-check constraints*, limit the values tuples may take, independently of any other tuples. For example, a domain constraint might specify that all values in the percentage field be between 0 and 100 inclusive, or that an employee's overtime_wage be at least 20% greater than his or her base_wage. (In Codd's original model, domain constraints are defined only over individual attributes, so the latter example would be a different kind of constraint in his terminology.)

A fundamental constraint in the relational model is *entity integrity*, which requires that all attributes that constitute the primary key for a relation be non-null. This constraint is necessary because of the requirement that the primary key attributes uniquely identify tuples; hence, none of those attributes are permitted to be missing.

Table-check constraints define invariants over individual relations. Uniqueness of primary keys is an example of a table-check constraint.

Relationship constraints control relationships among tuples in different relations. One type of constraint in this category that is central to the relational model is *referential integrity* [29, ch. 4]. A referential integrity constraint enforces a many-to-one mapping from a *referencing* relation to a *referenced* relation.

1. In this context, the word *integrity* refers to representation invariants that the data should satisfy. Regrettably, work on IFC uses the same word for a related but distinct concept, as described in chapter 3. This dissertation uses the term *integrity constraint*, or simply *constraint*, when the former meaning is intended. Clark and Wilson [22] compare the two types of integrity.

> Specifically, a set of columns in the referencing relation constitute a *foreign key*, which must match the primary key of some tuple in the referenced relation.[2]

In an information flow system, constraints that relate data with different labels are problematic because they can create covert channels. Additionally, it is desirable to create *classification constraints*, which restrict the labels themselves. Chapter 5 explores the issue, and presents IFDB's approach.

4.1.4 Data Independence and Views

Data independence is a key innovation in the relational model. It divorces the manner in which applications refer to data from the underlying representation of that data. Thus, changing the representation does not require modifying applications. *Physical* data independence protects applications from changes in the physical layout of the data; for instance, applications can use the same queries regardless of what indexes are available. *Logical* data independence permits the administrator to change the schema, perhaps by partitioning a relation or adding new fields, while maintaining compatibility with existing applications.

The view abstraction is an important part of the relational model's support for logical data independence. A view is a virtual relation, constructed by applying relational operators to base (physical) relations. An application can access data through the view, without regard for how the underlying relations are stored and indexed. This flexibility is often used to optimize the on-disk representation and the methods used to access the data. Views can be used to provide a restricted picture of a base relation, which makes them useful for security as well.

An important consideration in adding information flow extensions is ensuring that security concerns do not sacrifice data independence. Specifically, the manner in which data are labeled should be insulated from how applications access the data, and from how the data are stored on disk. Section 4.5 explains how Query by Label achieves this goal.

2. Relations may also refer to themselves. For example, a ClassProjects table with primary key student_id may have a partner_id field that refers to each student's project partner in the same table.

4.2 Labels in the Database

IFDB uses labels to track information flows within the database. Labels are attached to data at the granularity of tuples. The choice of granularity has important implications for the flexibility and efficiency of the system: why not label fields, tables, or entire databases? To address that question, this section reviews all four options and explains why tuple labels are the best choice for IFDB. The choices are presented in order of granularity, from coarsest to finest.

Per-database labels. As described in section 10.5.1, several systems designed for military use store different classes of information in different databases. The advantage of this approach is that it requires no trust in the DBMS. However, associating a single label pair with an entire database is impractical for DIFC; at a minimum, each user would require a separate database, and sharing would be problematic.

Per-table labels. In this design, each table has a static secrecy/integrity label pair, which applies to all tuples in the table. As section 10.2.3 explains, this approach is used by ASD_Views [55], while LDV [43] supports a variant: labels on columns. Like the preceding option, per-table labels do not match application needs. Database tables typically store data on behalf of many users, and each user may have distinct security concerns. Per-table labeling necessitates storing every user's data in a separate table, which makes *ad hoc* queries over many users' data tedious.

Per-tuple labels. This is the strategy IFDB employs. The intuitive justification for this model is that the data in each tuple are related to a single entity, for instance, a hospital patient. Hence, labels covering entire tuples are appropriate for many applications. Per-tuple labels represent a good trade-off between the ability to label data at a fine granularity and the space overhead associated with the labels. Figure 4-1 in the following section shows an example of labeled tuples.

Per-field labels. Several multi-level secure database systems, such as SeaView [101] and SINTRA [50], use per-field labels. However, this strategy adds substantial overhead, and the semantics are complicated, as explained in section 10.4.1. Furthermore, IFDB introduces declassifying views (section 4.5), which can be used to achieve the power of field-level labels.

In IFDB, secrecy and integrity labels for each tuple are stored in columns called _label and _ilabel, respectively. The leading underscore is intended to reduce the chances that the name conflicts with existing applications. The columns are hidden by default to avoid confusing applications that are not expecting them; they are only included in results of queries if they are requested explicitly. This approach follows existing practice: for example, the ROWID column in DB2 and the OID column in PostgreSQL are similarly hidden.

The database schema, which includes the definitions of all the tables, indexes, and views, is considered public. Any database user can read it, and only the administrator can write it, so in effect it has an empty secrecy label, and integrity label $\top$ (which contains all integrity tags). In contrast, some other database systems, such as SeaView, support the notion that parts of the schema – specifically, the very existence of certain tables – can have an associated secrecy level [37]. In a large organization using a single database, labeling the schema in this way makes it possible to have secret projects, whose existence is hidden from unauthorized users.

In fact, labeled schemas would not be difficult to implement; IFDB stores the schema definition in a set of special system tables, whose contents can be labeled. However, labeling schemas complicates the programming model: if the existence of certain tables is secret, then the code that generates queries on those tables must be secret as well. This line of reasoning adds a whole new, largely unexplored, dimension to DIFC, which is beyond the scope of this work. Like prior DIFC work, IFDB is intended to protect user data from misuse, not to protect the workings of the application from programmers who work on it. Therefore, secret projects can presently be supported only via separate databases and separate applications.

4.3 Queries

In a departure from the standard relational model, the semantics of a statement under Query by Label depend upon the label pair of the process that issued the statement. Specifically, the process labels affect which tuples the process is able to read and write. This section explains the basic model, but first, it motivates the need for a new model by illustrating why standard query semantics are inappropriate for an information flow system.

Inpatients

_label	patient_name	patient_dob	problem	condition
{alice_medical}	Alice	2/13/1960	cancer	good
{bob_medical}	Bob	6/26/1978	trauma	serious
{cathy_medical}	Cathy	4/22/1941	pneumonia	critical

Figure 4-1: The Inpatients table is an example of sensitive, labeled tuples stored in a database. It contains records about hospital inpatients, their problems on admission, and their general condition.

4.3.1 Information Flow in the Relational Model

Under the standard model, a query over a table conceptually reads every tuple in the table, then transforms and filters the results via a series of relational operators, such as SELECTs. The fact that tuples are selected by *content* has important implications for an information flow system: the set of results returned by a query reveals information about the content of tuples that were *not* returned by the query.

For example, suppose the table in figure 4-1 is stored in an SQL database for a medical information system. A _label column that denotes the secrecy of each tuple has been included as well; integrity labels are omitted for simplicity. Consider the following query:

```
SELECT name FROM PatientRecords WHERE problem <> 'cancer'
```

In standard ANSI SQL, the query produces the names of patients in a clinic who do not have cancer, namely, Bob and Cathy. One might think that the secrecy of the result could be captured by the label {bob_medical, cathy_medical}. However, to anyone who consults the public patient directory and finds out that Alice is a patient in the hospital, the results also reveal that Alice has cancer, because her tuple was excluded.

In fact, under the standard semantics, the correct secrecy label for the cancer query includes all three patients' medical tags. More generally, the contamination associated with a standard SQL query would necessarily include the greatest lower bound of the labels of every tuple in the table! This approach isn't practical; processes need a way limit their contamination because, as explained in chapter 3, they will be unable to communicate with the outside world if they are too contaminated.

4.3.2 Reads

In the Query by Label model, each query has a secrecy/integrity label pair, which corresponds to the labels of the process that issued the query. The model addresses the problem described in the preceding section by limiting the scope of the query to a subset of the database containing only the tuples whose labels are less restrictive than the labels of the query. This restriction is formalized as follows:

Rule 4.1. (Label Confinement)
A query performed by a process P with labels $(P.L_S, P.L_I)$ is performed on a subset of the database consisting of all tuples τ_i with labels $(\tau_i.L_S, \tau_i.L_I)$ such that $(\tau_i.L_S, \tau_i.L_I) \preceq (P.L_S, P.L_I)$.

For reads, the label confinement rule is simply an instantiation of the safe information flow rule (rule 3.1) given in section 3.2: A process should not see tuples whose contamination isn't covered by its own label. In one sense, the rule merely reflects the fact that label changes in the Aeolus model are explicit, but in another sense, the rule is fundamental to systems that query information by content. The cancer patient example in section 4.3.1 illustrates that it is necessary to limit the scope of queries in an information flow system.

Referring back to the Inpatients table shown in figure 4-1, consider the example again in the context of Query by Label:

```
SELECT name FROM PatientRecords WHERE problem <> 'cancer'
```

If a process issues this query with secrecy label {bob_medical} and an empty integrity label, the results will include only Bob's records. The result conveys nothing about the other patients because the process's view of the database is confined to only tuples whose labels are a subset of {bob_medical}. Thus, the process label always reflects the contamination of all the data that might have influenced the process.

4.3.3 Explicit Query by Label

Query by Label limits queries to tuples whose labels are subsets of the process label. However, processes can specify additional conditions on the label explicitly by referring to the _label and _ilabel columns. As with all queries, the scope is limited to tuples with labels no more restrictive than the process label; any additional conditions further limit the results. For example, the following SQL query selects all meetings whose secrecy labels contain the programX tag.

```
SELECT * FROM Meetings WHERE _label @> '{programX}'
```

The @> operator in the query implements the mathematical superset operator $\supseteq$ for labels. Of course, the process that issues the query must have at least the tag programX in its label, or the query will return no results!

Queries such as this one are not commonly needed. For instance, it would be simpler to add a meeting_purpose field to the Meetings table and use that to determine which meetings are about Program X, rather than ascribing additional meaning (specifically, the purpose of the meeting) to tags in the secrecy label. However, explicit Query by Label will be revisited in section 5.2, where it is used as a way to limit the effects of polyinstantiation.

4.3.4 Writes

Writes to the database in Query by Label are more restricted than reads: the label of the process must match the tuple being written exactly. This requirement stems from the combination of two separate stipulations:

W1. *Tuples a process writes must reflect the contamination of everything the process has read.* Safe information flow (rule 3.1) requires that information can flow from a source (such as process P) to a destination (such as tuple τ) only if $(P.L_S, P.L_I) \preceq (\tau.L_S, \tau.L_I)$. In other words, a process cannot write a tuple with a lower label pair than itself.

W2. *Writes should not affect tuples the process is unable to see.* Label confinement (rule 4.1) requires that a statement issued by a process P apply to a subset of the database consisting of tuples τ_i such that $(\tau_i.L_S, \tau_i.L_I) \preceq (P.L_S, P.L_I)$. That is, the effect of the statement is limited to tuples with label pairs no higher than the process labels.

Formally, the conjunction of these requirements gives the following rule for writes:

Rule 4.2. (Write Rule)
A process with labels $(P.L_S, P.L_I)$ can write a tuple with label $(\tau.L_S, \tau.L_I)$ only if $P.L_S = \tau.L_S$ and $P.L_I = \tau.L_I$.

In other words, all writes have exactly the label of the process. Traditionally, requirement W1 is called the $\star$ property, while rule 4.2 is called the strong-$\star$ property (see section 10.1).

Since processes see only a subset of the database, it is possible that a write might violate an integrity constraint, even though it looks okay to the process. For example, a process might attempt to insert a tuple with the same primary key as an existing tuple that it is unable to see. Chapter 5 discusses how IFDB addresses this problem.

A natural question about rule 4.2 is whether it is overly restrictive. The rule comes from two requirements: W1 and W2. Requirement W1 is a standard prerequisite for safe information flow, but requirement W2 bears some explanation. After all, W2 is typically the rule that is applied to reads, not writes. Indeed, prior work that develops DIFC semantics for file systems [18, 106, 152] does not impose this restriction. In a database, however, allowing a process to write a tuple with a more restrictive label introduces a covert channel: it allows the process to learn about the tuple, which is more contaminated. Essentially, the problem is that *writes in a DBMS are never blind; each write is also a read.*

Lifting this restriction by supporting blind writes to the database is conceivable. However, the resulting semantics are so awkward that the option seems unrealistic. The following are some of the problems that can arise if processes were allowed to write tuples they cannot see:

— Allowing processes to write tuples they cannot see can lead to potentially dangerous confusion. For example, a process might convert Bob's medical records to a new format, store them in a separate table, and delete the original records. However, if Bob has records with more restrictive labels than the process is allowed to see,[3] the process might delete records that have not been converted.

— Processes would be able to learn how many tuples (if any) were affected by an update. For example, consider the following SQL statement on the table in figure 4-1, executed with an empty secrecy label:

```
UPDATE PatientRecords SET name = 'Alice'
    WHERE name = 'Alice' AND problem = 'cancer'
```

The statement has no effect, but it returns the number of rows updated: one if Alice has cancer, and zero otherwise. Information flow requirements necessitate suppressing this count. Hence, applications cannot know the effects of their updates, or whether the updates succeeded at all.

3. In the United States, for instance, mental health records are subject to more stringent secrecy requirements than other types of medical records. [1]

— The requested write might violate an integrity constraint. Mitigating actions the system might take, such as reporting the error or aborting the current transaction, are observable to the application; therefore, they also leak information about the tuples written. (Integrity constraints are discussed further in chapter 5.)

— The write might lead to conflicts with concurrent operations running on behalf of other processes that have different labels. These types of conflicts can also leak information, as discussed further in chapter 6.

A consequence of the write rule is that the label associated with each tuple is constant. In fact, the restriction that data labels do not change is even more fundamental, and is present in other information flow systems as well. Raising the label of a tuple would remove if from the purview of clients running with the old label. To those clients, the tuple appears to have been deleted. Therefore, if a client wants to reclassify a tuple with a different label, it must do so by deleting the tuple and inserting a new one. Of course, to do this, the process must be able to do two writes with different labels. The following section explains how processes can write with different labels by changing their labels.

4.3.5 Writing Tuples With Different Labels

Applications need the ability to write tuples with different labels as part of a single logical action. For example, a process that creates an account for a doctor in a medical clinic might write two tuples: one with the doctor's contact information, and another containing his password. Rule 4.2 would seem to prohibit that kind of operation, since it requires that a process always writes tuples with exactly its own secrecy and integrity labels.

Given that it is problematic to allow processes to write tuples with labels other than the current process label, how does a process write tuples with different labels? The answer is that processes can change their labels as explained in chapter 3. It is always safe for a process to raise its label, making it more restrictive. Making a label more restrictive entails adding secrecy tags or removing integrity tags. Furthermore, if the process has authority for a tag, it can declassify that tag, removing the tag from the secrecy label, or it can endorse the tag, adding it to the integrity label. Returning to the example of adding a doctor's account, an application could simply write the doctor's contact information, change its label, and then write the doctor's password.

The Query by Label semantics of SQL queries issued by a process with the label pair $(P.L_S, P.L_I)$ are summarized below.

SELECT queries operate over a subset of the database containing only the tuples whose labels are less restrictive than the process secrecy/integrity label pair. Formally, if $\mathcal{D}$ is the set of tuples in the database, then the result of the query is equivalent to that of a standard SQL query on database $\mathcal{D}'$, where $\mathcal{D}' = \{\tau | \tau \in \mathcal{D} \wedge (\tau.L_S, \tau.L_I) \preceq (P.L_S, P.L_I)\}$.

Query by Label does not change the relational algebra, so complex operations such as joins and aggregate functions have the usual semantics. However, they operate over relations that contain only the tuples the process is able to see given its labels.

INSERT statements add tuples with exactly the label pair of the process. The INSERT … SELECT form, which inserts tuples that were produced by a query, also writes the tuples with the secrecy and integrity labels of the process, even if the source tuples have less restrictive labels.

UPDATE statements affect only tuples with the same label pair as the process. Tuples with more restrictive labels are not visible to the process and are ignored. Tuples with less restrictive labels can't be written by the process without violating the information flow rules; attempting to update such tuples is an error.

DELETE statements affect only tuples with the same label pair as the process. As with UPDATE statements, tuples with more restrictive labels are unaffected, and attempting to delete tuples with less restrictive labels is an error.

Nested queries, including the SELECT clause of an INSERT … SELECT query, are handled as implied by the semantics of the basic operations. Data Definition Language (DDL) commands, which modify the schema, are restricted to the administrator. The semantics of Transaction Control Language (TCL) statements are covered in chapter 6.

Figure 4-2: Semantics of basic SQL statements under Query by Label

Allowing processes to change their labels to write tuples with different labels actually results in the *same* information leaks as described previously. However, it provides a principled way of reasoning about these leaks. Specifically, the Query by Label model supports the principle articulated in the introduction to this chapter: unsafe information flows must be accompanied by an explicit exercise of authority.

Presenting a simple model to programmers is important because covert channels are notoriously difficult to reason about [108]. In fact, the rule for writes presented in this section isn't the whole story. As chapters 5 and 6 point out, interactions between IFC and other DBMS features, such as constraints and transactions, give rise to new issues. Thinking about multi-level writes in terms of simple writes and label changes is key to providing intuitive semantics for these features. Sections 4.4 and 4.5 present abstractions that provide more convenient ways to do writes, but importantly, these abstractions can be understood in terms of label changes in the basic model.

4.4 Application Code in the DBMS

In designing a database-backed application, an important consideration is how to partition the computation between the application and the DBMS. Performing computations in the database system instead of the application has several widely recognized benefits. For one, it improves performance: less data must be transmitted between the application and the DBMS, and transaction latencies are lower. Also, performing computations in the DBMS via stored procedures improves modularity and security by providing a narrow, well-defined interface between the application and the database. A variety of procedural extensions to SQL, such as PL/SQL [46], PL/pgSQL [56], and SQL/PSM [3] have been developed to support imperative programming in the DBMS.

Chapter 3 explains that applications use the Aeolus API (figure 3-1) to manage information flows. Computations running within the DBMS require this ability as well, so IFDB implements the Aeolus API as a set of primitive stored procedures that can be invoked within SQL statements and stored procedures. Stored procedures start running with the labels and authority of the process that invoked them. Any label changes the procedure makes (for instance, adding a secrecy tag or declassifying) are reflected in the labels of the calling process.

IFDB aims to provide a uniform API, so that the same operations are available in both SQL stored procedures and in the language the application is written in (such

as PHP-IF and Python-IF, described in chapter 2). To that end, IFDB also supports *stored authority closures*, which extend the authority closures described in section 3.5 to the DBMS. Stored authority closures bind special authority to stored procedures. They are important because they allow application programmers to add functions to the database that perform operations that would otherwise not be allowed by the information flow rules.

For example, the programmer might want to use a userlist tag to protect the list of all usernames registered with the application, so that bugs in the application do not allow an attacker to see the list. However, the code that registers a new user account must still be able to check if a given username is already in use. Reporting whether a requested username is taken represents a small information leak, so IFDB requires an exercise of authority, namely a declassification, to vouch that this leak is acceptable. The code can be put in a stored authority closure with the appropriate authority, as in the following example. The example is written in IFDB's extended version of PL/pgSQL.

```
CREATE FUNCTION addUser(uname VARCHAR)
RETURNS VOID PRINCIPAL accountManager AS $$
BEGIN
        PERFORM addSecrecy(userlist);
        INSERT INTO Users (username) VALUES (uname);
        PERFORM declassify(userlist);
        EXCEPTION WHEN OTHERS THEN      -- catch any exception
                PERFORM declassify(userlist);
                RAISE;                  -- re-throw the exception
END;
$$ LANGUAGE plpgsql;
```

The circled PRINCIPAL *accountManager* clause identifies the stored procedure as an authority closure, which runs with the authority of the accountManager principal when it is invoked. The closure adds the userlist tag to the secrecy label of the client process, inserts the username into the Users table, and declassifies to remove the tag it added. (The italicized identifiers *accountManager* and *userlist* are placeholders for the appropriate principal and tag IDs. Prior to creating the closure, the administrator must have previously created the accountManager principal and userlist tag.)

Upon return from an authority closure, the secrecy and integrity labels of the process are merged with the caller's original labels, as in Aeolus: the secrecy labels are

unioned, and the integrity labels are intersected. Doing this prevents closures from removing the caller's contamination, thus preventing confused deputy problems. For example, it prevents a process that is already contaminated with the userlist tag from abusing the addUser closure to remove that contamination.

The example also illustrates an important point about exceptions. Suppose the username is already present in the table and properly labeled. In this case, the INSERT statement will fail due to a uniqueness constraint violation and the transaction will abort. When such an error occurs inside an authority closure, IFDB restores the process principal to that of the caller. However, as in Aeolus, the process labels retain the contamination of anything the authority closure read, which ensures that authority closures cannot leak information. (If the labels were restored in the event of an exception, a closure could leak bit of a secret it has no authority to declassify by reading the secret and deciding whether to raise an exception based on the value it read.) In this particular example, the constraint violation is an anticipated possibility, so the closure catches the exception, uses its authority to declassify the userlist tag, and re-throws the exception.

4.5 Declassifying and Endorsing Views

The preceding section introduced stored authority closures, which extend the stored procedure abstraction. However, relational databases typically make use of a different abstraction – views. The Query by Label model provides new mechanisms, declassifying and endorsing views, which extend the view abstraction analogously. This section describes how declassifying and endorsing views work, and shows that they are instrumental in preserving data independence.

4.5.1 Defining Views Using Authority

IFDB extends the Aeolus model with declassifying and endorsing views, which are adaptations of authority closures to the relational model. Instead of binding authority to code, the authority is bound to the definition of a view, which uses that authority to declassify or endorse. Declassifying and endorsing views thus provide a means of expressing *downgrading policies* [20, 93] as relations; they allow information with high labels to be accessed with lower labels through the view, after that information has been appropriately sanitized.

For example, patient privacy regulations in the United States permit hospitals to publish a directory of inpatients and their general conditions [1]. A hospital could produce this information with a declassifying view of the Inpatients table (figure 4-1) as follows, with the new syntax circled:

```
CREATE VIEW PatientDirectory AS
    SELECT patient_name, condition FROM Inpatients
    WITH DECLASSIFYING (all_patients_medical);
```

The view has authority to declassify for any tags specified in the DECLASSIFYING clause, and the creator of the view must have this authority as well. In this case, the view has authority for the all_patients_medical compound tag, and it uses its authority to declassify the patient medical tags in the base relation, which are all members of this compound tag. Endorsing views work similarly, but for integrity; for instance, an endorsing view might add an integrity tag to each tuple after passing it through a sanitization or verification function.

The PatientDirectory view is a simple example because all the tuples visible in the view have the same label. However, some downgrading policies may require more expressive power than a DECLASSIFYING or ENDORSING clause can provide. Suppose the hospital wishes to properly track information flows in the patient billing application. Billing records and medical records may have different confidentiality policies; billing records are sent to insurance companies for claim processing, but medical records are not. A view that extracts medical procedure codes from patient medical histories to produce bills for those procedures might take a tuple with secrecy label {alice_medical} and produce a tuple labeled {alice_billing}. Thus, each patient's data is declassified (and reclassified) differently. Query by Label supports this type of view via RELABEL clauses:

```
CREATE VIEW PatientBills AS
    SELECT patient_id, cost(med_procedure) FROM PatientMedProcedures
    WITH RELABEL billing_to_medical PRINCIPAL allPatients;
```

The RELABEL clause names the billing_to_medical relabeling function, which takes a secrecy/integrity label pair as input and produces a new label pair. The function is invoked with the labels of the process that is querying the view, and it produces the appropriate labels for a query on the underlying relation. In this case, if the secrecy label contains alice_billing, the function adds the associated alice_medical tag. It must look up the correspondence between the patient billing and medical tags in a separate table. Conceptually, the contents of the view as seen by a given

process are produced by invoking the relabeling function and updating the process labels as it specifies, running the query in the view definition, and then restoring the original labels of the process.

The PRINCIPAL allPatients clause specifies the authority that the view requires to do its job. In effect, each query over the view involves three steps:

1. Add the patient's medical tag to the secrecy label of the process, and declassify the patient's billing tag. The billing_to_medical provides the appropriate mapping between the labels.

2. Execute the query specified by the view definition, which extracts tuples from the underlying PatientMedProcedures table.

3. Reverse the first step, adding the patient's billing tag and removing the patient's medical tag from the secrecy label of the process.

Since the view effectively declassifies arbitrary patients' medical and billing tags, it uses authority for allPatients – a principal that acts for all patients. Before executing queries over the view, the DBMS checks that this principal has authority for any tags that the billing_to_medical function requested to change. The creator of the view must acts for the principal specified in the PRINCIPAL clause.

Relabeling views are more general than views using DECLASSIFYING and ENDORSING clauses. The PatientDirectory view defined earlier could be defined as a relabeling view, where the relabel function adds the all_patients_medical tag to the secrecy label of the process. However, relabeling views are implemented differently, and are subject to a minor restriction. Specifically, queries that invoke stored procedures that may have side-effects cannot be run over relabeling views, as explained in section 8.1.3.

4.5.2 Discretionary Views

The examples presented in the preceding section are views that define mandatory policies that affect all users. Chapter 3 touted discretionary security, where principals can define the policies for their own data, as a benefit of the Aeolus model. Thus, individual principals, not just the administrator, should be able to create declassifying and endorsing views. For example, in a social network, each user could define a view, vested with his own authority, that determines what personal information about the user is available to his friends.

However, views are part of the database schema, and as explained in section 4.2, only the administrator may modify the schema. To allow non-administrators to create views, each principal is given a *private schema*, and can create tables in that schema. Such tables are referred to as *schema name* . *table name*, where the schema name is based on the ID of the principal. Thus, the names don't conflict with views of other users, or with tables and views in the global schema, which is managed by the administrator. To ensure that schema modifications themselves cannot be used to signal sensitive information, a process may only create, modify, or delete a view in its private schema if the process has an empty secrecy label.

4.5.3 View Updates

Since views contain derived data, insert, update, and delete operations on them are not always well-defined. The IFDB implementation is based on PostgreSQL, and as such, it leverages PostgreSQL's solution to the view update problem. PostgreSQL requires the database programmer to create rewrite rules describing how to transform inserts, updates and deletes on a view into changes to the underlying tables. For example, suppose that patient contact and billing information are stored in Contact and Billing tables, and a ContactBilling view has been defined as an equi-join of the two tables. An insert into the ContactBilling view could thus be rewritten as two inserts: one into Contact and the other into Billing. In IFDB, rewrite rules for declassifying and endorsing views may need to invoke stored authority closures to write base tuples with different labels than the process. In the ContactBilling example, Contact tuples might have different classifications than Billing tuples, and thus authority is needed to change the process label appropriately for each insert.

IFDB also modifies the meaning of the SQL standard WITH CHECK OPTION clause, which is specified in a view definition. The clause normally instructs the DBMS to enforce a correctness condition on insert and update rules: following a successful modification, the inserted or updated tuple must actually exist in the view. For example, the option ensures that an attempt to insert a living patient into a view that selects only dead patients from a table containing all patients is an error. When the check option is specified in IFDB, IFDB enforces a stronger property, which constrains the label pair of the tuple: the tuple must exist in the view *and* be visible to the process that inserted it.

Views that do *not* use WITH CHECK OPTION are particularly useful in IFDB. They provide a convenient way to support updates that transform data with higher

labels than the requesting process. As described in section 4.3.4, all writes to the database in the Query by Label model are restricted to tuples with exactly the labels of the process. However, this rule is often too constraining. For example, a common requirement in password-based authentication systems is that users should not be able to see their own passwords, but they should be able to change their passwords. An update rule on the Passwords table that invokes an authority closure provides the needed power with familiar syntax: users can simply update their own entries in the Passwords table, even if they lack the ability to see their passwords. As section 4.3.5 explains, writing tuples with higher labels can leak information, so the authority closure underlying the update rule must exercise appropriate authority to raise the process labels, perform the write, and declassify. By declassifying, it vouches that the disclosure (namely, whether the update succeeded) is acceptable.

ANSI SQL specifies an alternative to rule-based view writes: certain types of views for which updates have an unambiguous interpretation should automatically be updatable. Neither PostgreSQL nor IFDB presently support this feature, but many commercial database systems do, and support is planned for a future PostgreSQL release. In IFDB, these updatable views will require relabeling rules, similar to reads (see the preceding section).

4.5.4 Data Independence

Since data are labeled at the granularity of tuples in IFDB, how is it possible to handle cases where a single logical entity has attributes that should have different labels? One solution is to force the database designer to partition the entity into multiple tuples according to the confidentiality and integrity concerns, but this approach violates the data independence principle introduced in section 4.1.4. In the medical system, for example, patients' billing information and contact information have distinct labels and security policies. Nevertheless, the data may be accessed together frequently, such as when generating medical bills. The principle of data independence dictates that the programmer should be able to refer to the billing and contact information as separate relations, a single relation, or some combination thereof, regardless of the underlying representation.

IFDB achieves data independence through a combination of standard views and declassifying views. If the underlying representation is a single ContactBilling table, as shown in figure 4-3(a), then the Contact and Billing relations shown in figure 4-3(b) can be generated as declassifying views that project the appropriate

ContactBilling

_label	name	address	cardnum
{alice_contact, alice_billing}	Alice	32 Vassar St	12345678
{bob_contact, bob_billing}	Bob	52 Brattle St	90274963
{cathy_contact, cathy_billing}	Cathy	497 Boylston St	29349274

(a) Since contact and billing information are frequently accessed together (for example, when printing a bill), they can be stored in a single table. The labels of these combined tuples must contain both the contact and billing tags.

Contact

_label	name	address
{alice_contact}	Alice	32 Vassar St
{bob_contact}	Bob	52 Brattle St
{cathy_contact}	Cathy	497 Boylston St

Billing

_label	name	cardnum
{alice_billing}	Alice	12345678
{bob_billing}	Bob	90274963
{cathy_billing}	Cathy	29349274

(b) Separate Contact and Billing relations reflect the distinct confidentiality concerns of contact and billing information. They can be produced as declassifying views of the ContactBilling base table, or alternatively, ContactBilling can be defined as a view that joins the Contact and Billing base relations.

ContactBilling

_label	name	address	cardnum
{alice_contact}	Alice	32 Vassar St	*NULL*

(c) If ContactBilling is defined as a view using a full outer join, a process with secrecy label {alice_contact} will see null values for higher-labeled fields, similar to MLS database systems with field-granularity classification.

Figure 4-3: Ordinary and declassifying views make it possible to keep the representation of data independent from its security requirements.

columns. Conversely, if the underlying representation of the data is as two separate tables, the merged ContactBilling view can be produced as an ordinary view that joins the two base tables.

If the ContactBilling table is constructed as a view, there are several options. The view can be constructed as an inner join, in which case a tuple for a patient will not be visible unless the process label includes both the patient's billing and contact tags. Alternatively, the view can be constructed as a full outer join, in which case processes will simply see null values in place of the fields that its label does not allow it to see, as illustrated by figure 4-3(c). The latter option can be used to simulate field-level labels, with semantics similar to the SeaView model [101]. Thus, the use of row-granularity labels in IFDB is not a significant limitation; finer-grained labeling can be achieved through views when needed.

4.6 Access Control and Clearance

This chapter has introduced the Query by Label model as an effective way to secure databases using decentralized information flow control. As the introduction argued, DIFC has advantages over access control: it uses sandboxing to allow untrusted code to compute on sensitive data safely, and it enforces end-to-end security policies.

However, this dissertation does not propose to completely supplant access control. There are some security policies that can be enforced more directly with access control. Consider the example of the Passwords table introduced in section 4.5.3. This table has a secrecy policy that says that only the code that authenticates Alice should be able to read Alice's password, and an integrity policy that only Alice should be able to write Alice's password. Although DIFC enables more sophisticated policies that allow untrusted code to compute over the passwords without being able to release them, untrusted code in fact has no business reading Alice's password. Therefore, fine-grained access control is a more fitting way to secure the contents of the Passwords table. Access control also avoids some of the covert channels that are problematic in information flow systems.

A particular type of access control called *clearance* meshes well with the DIFC model. Clearance restricts the maximum secrecy label and the minimum integrity label a process is allowed to have. For example, if Alice's password tuple has the alice_password secrecy tag, then processes that are not cleared to add alice_password to their labels cannot read Alice's password. Clearance has been studied in the

Asbestos [141] and HiStar [152] operating systems, and other fine-grained access control techniques for databases have also been developed (see section 10.3). IFDB includes the forms of access control built in to PostgreSQL, including ANSI SQL role-based access control and view-based access control; access control based on clearance would be a useful extension. A limited form of clearance is introduced in section 6.2 to prevent unsafe flows between conflicting transactions.

Chapter 5 Constraints

Databases provide means to enforce integrity constraints, which are invariants that data elements and groups of related data elements must conform to. It is highly desirable to enforce integrity constraints in the database, since doing so suppresses a large class of application errors.

Unfortunately, dynamic constraint enforcement raises problems when information flow rules prevent processes from being influenced by higher-labeled data. For example, suppose a medical application running with an empty secrecy label attempts to insert a tuple for Bob into the Patients table, but an entry with the same primary key and a higher secrecy label already exists. The two obvious approaches are raising an error, which leaks information by apprising the application of the existence of the secret tuple, or allowing the constraint to be violated. Neither alternative is entirely satisfactory.

Many real-world applications, such as Wikipedia, avoid database-enforced constraints due to the expense involved. Followers of that school of thought might propose abdicating the problem of constraint enforcement to the application. However, rather than solve the problem, this approach merely shifts responsibility and increases the likelihood of mistakes. For instance, suppose that every patient appointment record in a medical information system has an associated patient visit record containing medical notes from the visit. Further suppose that nurses should not be able to view visit records associated with the mental health clinic. Unfortunately, nurses might be able to infer that a patient has visited the mental health clinic if

he has an appointment record without a visible visit record. The problem exists regardless of whether the database or the application enforces the natural constraint between appointments and visit records.

This is an example of the *inference problem*: it is possible to infer something about a secret by looking at related data that is less secret. These inferences often involve unstated assumptions or outside knowledge [102], so it is impossible for IFDB to anticipate them. However, it is possible to prevent inferences for constraints that are explicitly stated and enforced in the DBMS. IFDB provides a semantics for constraint enforcement that ensures that constraints do not leak information, except via an explicit use of authority (declassification or endorsement). This secure-by-default design provides a framework for making principled decision about how sensitive information is used, and how it might be exposed.

This chapter discusses IFDB's approach for handling constraints in the presence of information flow restrictions. In the relational model, constraints can be broken down into four categories: domain constraints, which concern individual tuples; table-check constraints, which relate tuples within a table; relationship constraints, which relate tuples in different tables; and more general kinds of constraints. The following sections treat each of these categories individually, and section 5.5 introduces constraints on information flow labels themselves. Refer back to section 4.1.3 for a summary of how these constraints are interpreted in an ordinary relational database.

5.1 Domain Constraints

The simplest constraints are domain constraints, which restrict the values tuples may take, independent of any other tuples. Domain constraints don't introduce any information flow issues in IFDB. The constraints themselves are part of the database schema, which is considered public (see section 4.2). Applications can evaluate for themselves whether particular tuples satisfy domain constraints, and therefore the DBMS doesn't convey any new information by enforcing the constraint.

However, domain constraints would be problematic if IFDB's write rule, given in section 4.3.4, did not prevent applications from updating tuples they cannot see. For example, suppose there is a constraint that an employee's overtime_wage be at least 20% greater than his or her base_wage. An application could learn the salary of an employee with a high secrecy label by issuing updates and observing which values of overtime_wage are permitted.

Domain constraints can also introduce complications in database systems such as SeaView [101], in which different fields within a tuple can have different labels. For instance, if a tuple's base_wage had a more restrictive label than its overtime_wage, it would be possible to learn the value of the former field by decreasing the latter until a constraint violation is observed.

5.2 Table-Check Constraints

Table-check constraints relate tuples in a single table. This section focuses on uniqueness constraints, which are the most common type of table-check constraint, and the only kind supported directly by the ANSI SQL standard. Other types of table-check constraints are uncommon. An example is PostgreSQL's exclusion constraints, which generalize uniqueness constraints to permit specifications such as "No two salespeople can be assigned to locations within 100 miles of each other." The issues that arise with this kind of constraint are analogous to the problems with uniqueness constraints. Other table-check constraints that don't follow this pattern can be handled as described in section 5.4.

5.2.1 The Problem with Uniqueness Constraints

It is easy to check that a write obeys a constraint when all of the tuples that must be read to verify satisfaction of the constraint are visible to the process performing the write. However, a problem arises when the very question of whether the data conforms to the constraint depends on tuples the process should not see.

The table in figure 5-1 will be used to construct an example of the problem. Integrity labels have been omitted to simplify the presentation. Consider the following inserts into the table:

1. Insert (Dan, 8/12/1969, 2B) into HIVPatients with any label.
2. Insert (Alice, 2/1/1960, 2A) into HIVPatients with secrecy label {alice_medical}.
3. Insert (Alice, 2/1/1960, 2A) into HIVPatients with secrecy label {}.

The first insert doesn't violate the constraint because there is no entry for Dan in the table; hence, it should succeed regardless of the label used. The second insert violates the constraint because Alice already has an entry in the table. Furthermore,

HIVPatients

_label	patient_name	patient_dob	virus_type
{alice_medical}	Alice	2/13/1960	1M
{bob_medical}	Bob	6/26/1978	1M
{cathy_medical}	Cathy	4/22/1941	1O

Primary key (patient_name, patient_dob)

Figure 5-1: In a medical records system, the HIVPatients table contains specialized records for patients with HIV.

enforcing the constraint and causing the second insert to fail reveals nothing, because the conflicting tuple is already visible to the process performing the insert. The third insert is the problematic one. Like the second insert, it violates the constraint; however, the process performing the insert has an empty secrecy label, so it isn't supposed to see the conflicting tuple, which has a higher secrecy label. Disallowing the insert would leak the fact that a tuple for Alice already exists. Anyone who knows Alice's name and date of birth could thus learn whether she has HIV by attempting this insert. Since the insert is performed by a process with an empty secrecy label, the process will be allowed to release the information to the outside world.

Uniqueness constraints pose a related problem for integrity. Processes with low integrity can insert tuples that subsequently interfere with the operation of processes with higher integrity. Thus, the high-integrity process could become confused or be unable to do its job because the uniqueness constraint prevents it from inserting. Definition 3.1 can be used to capture the secrecy and integrity problems in a single statement: a problem arises when a process attempts an insert that conflicts with a tuple with a more restrictive (higher) label pair.

Furthermore, the problem is not limited to just inserts. Updates can cause the same problems when a unique key value is updated such that it conflicts with the key of another tuple. To simplify the exposition, update operations are regarded as a delete of the old tuple followed by an insert of the new tuple. Therefore, the rest of section 5.2 discusses only inserts.

5.2.2 IFDB's Solution

IFDB uses *polyinstantiation*, which permits inserts of tuples that conflict with higher-labeled tuples. Clients running with lower labels, unaware of the higher-labeled

tuples, see a consistent view of the database and are unaffected by the higher-labeled data. Clients running with higher labels, however, will see both tuples, distinguished only by their labels – a violation of the uniqueness constraint.

The trouble with polyinstantiation is that it can lead to confusion. What does it mean to have two HIV patients with identical primary keys but different labels? Much prior work is concerned with the use of polyinstantiation as an important feature in its own right, and proposes *cover stories* as one possible answer to this question. Specifically, the public version of a patient record might be different from the secret version due to deliberate subterfuge. Polyinstantiation can also be used to express different subjective opinions about the truth. Section 10.4.1 reviews these proposals and the complexities they involve.

Although IFDB is capable of supporting these perspectives, this dissertation advocates a simpler interpretation, whereby polyinstantiated tuples are seen as mistakes. Since it would leak information to expose the mistakes to clients with lower labels (that is, by notifying them of the conflict), IFDB instead exposes the mistakes to the clients with higher labels.

As a result of this philosophy, IFDB only permits polyinstantiation when it is necessary to avoid covert channels. For example, consider again the third example from section 5.2.1: inserting (Alice, 2/1/1960, 2A) into HIVPatients with secrecy label {}. This insert causes polyinstantiation and results in two entries for Alice: a correctly labeled one with virus type 1M, and the new one with virus type 2A. However, if the same insert were performed with the equally bogus secrecy label {alice_medical, eve_medical}, the result would be an error. There is no need to polyinstantiate in the latter case because the process running with secrecy label {alice_medical, eve_medical} is able to see the conflicting tuple, which has secrecy label {alice_medical}; therefore, apprising the process of its mistake does not reveal any additional information to it. Unlike IFDB, many prior systems polyinstantiate in both cases for the sake of cover stories, calling the first example invisible polyinstantiation, and the second visible polyinstantiation [75].

Polyinstantiation means that queries could return multiple records when only one is expected. Some application code may not be prepared to cope with this eventuality. Therefore, IFDB has four ways to shield applications from the effects of polyinstantiation:

1. *Label constraints* that relate the unique key to the information flow label can prevent polyinstantiation. Label constraints are described in section 5.5.

2. The information flow labels can be made part of the key. This solution also prevents polyinstantiation and is applicable when cover stories or multiple versions of the truth are desired. For example, the Facebook website presently leaks information about users' private photos by caching a count of the number of photos they have in each album; Alice's employer might be able to see that Alice has 300 photos in the *Partying* album, even if the employer is only authorized to see two of the photos. To solve the problem, the system might store one count for Alice's public photos and one count for her private photos, both of which could be updated by triggers. Since storing multiple counts with different secrecy labels is intentional, the database designer should make the decision explicit by including the secrecy label in the key.

3. Queries can specify exact secrecy and integrity labels as the selection predicate for any relation. Since polyinstantiated tuples are guaranteed to have different labels, specifying exact labels guarantees that violation of uniqueness constraints will not be observed. This technique works well if the application knows the exact labels the data ought to have; the system returns the desired tuple and ignores the others, which are presumably the erroneous ones since they have the wrong labels.

4. To limit the effects of polyinstantiation due to incorrect secrecy labels, applications can use integrity labels. Integrity labels don't guarantee that no polyinstantiation will be observed, but they limit which code and which principals can cause confusion. For example, a process running with integrity label {alice_medical} will only observe polyinstantiated tuples if they were created by processes that used authority to endorse for the alice_medical tag. Presumably authority for this tag is limited to entities Alice trusts, such as her doctors, and if they are trustworthy they will apply the correct secrecy label.

The first two approaches are preferable because they actually prevent polyinstantiation rather than hide it. The latter two solutions are useful, however, if the correct labels aren't known *a priori*, or if label constraints are considered too expensive to enforce. (Generally speaking, label constraints are no more expensive than foreign key constraints.)

5.3 Relationship Constraints

Invariants can also be enforced between tables. The most common multi-table property in the relational model is referential integrity,[1] which enforces a many-to-one mapping between a *referencing* relation and a *referenced* relation. For example, for each patient visit record in a medical clinic, there should be a corresponding record with the patient's basic information, such as name and date of birth. In SQL, these constraints are called *foreign key constraints*, and they are enforced directly by adding FOREIGN KEY clauses to table definitions. This section addresses only foreign keys; IFDB handles other types of constraints with triggers, covered in section 5.4.

5.3.1 The Problems with Referential Constraints

Referential constraints create information channels when the referencing tuple and the referenced tuple have different labels. This section describes the problem for secrecy; there are analogous problems for integrity (see section 5.3.3), but they are not as serious. Suppose tuples in table A are constrained to reference tuples in table B. There are two problematic cases:

— *Inserts.* A process with any label can try to insert tuples into A in order to determine which tuples exist in B. For instance, suppose that in a medical system, every tuple in the HIVRecords table is constrained to refer to a patient listed in the HIVPatients table in figure 5-1. A process running with an empty secrecy label could learn whether a particular patient is in HIVPatients by trying to insert a tuple for that patient into HIVRecords. The insert will succeed only if the referenced patient has HIV.

  Moreover, the very existence of tuples in A reveals to any readers that the corresponding B tuples exist. The leak is established when the tuples are inserted into A in the first place, so this issue is in fact a consequence of the problem with inserts. A further concern is one of semantics: even though the constraint might be enforced over the entire database, processes may encounter dangling references because their views of the database are restricted.

— *Deletes.* Suppose that a tuple a in A has a label pair that is more restrictive than that of a tuple b in B, that is, $(a.L_S, a.L_I) \succ (b.L_S, b.L_I)$. Then a process

1. The use of the word *integrity* in *referential integrity* should not be confused with integrity tags. Also see footnote 1 on page 40.

P with label pair $(P.L_S, P.L_I) = (b.L_S, b.L_I)$ could learn that a exists by trying to delete b. For instance, in the preceding example, the HIVPatients table itself might refer to another table, PatientContact. If the DBMS disallowed deletion of a patient's contact information only if he has a referring tuple in HIVPatients, that provides an effective (albeit destructive) means to determine which patients have HIV. For instance, a process running with the same label as Alice's PatientContact tuple could attempt to delete that tuple, and if the delete fails due to a constraint violation, the process knows Alice has HIV.

Some multi-level-secure databases have attempted to address these problems by requiring the referencing and referenced tuples to have identical labels [72, 73, 101, 128]. This approach is overly limiting: for example, it disallows the constraint that each user in a Users table must have a password in the Passwords table (with a more restrictive label pair).

Other database systems require that the referencing tuples have labels at least as restrictive as the referenced tuples [13, 138]. This restriction solves the problem with inserts, and it ensures that processes don't observe any dangling references [52]. As with the preceding proposal, however, it disallows constraints such as the one requiring each user to have a password. Furthermore, it does not prevent the problem where deletes leak information.

One could imagine clever ways of handling deletes to avoid the channel. For instance, SQL constraints have an ON DELETE CASCADE option, which automatically deletes referencing tuples when their referents are removed. However, cascading deletes can still fail, and thereby leak information. This happens, for instance, when deleting one of the referencing tuples causes a violation of a different constraint. More critically, recursive deletes may run afoul of Query by Label's write rule (rule 4.2), in which case they will fail anyway.

5.3.2 IFDB's Solution

IFDB addresses both the insert and delete problems at the time of the insert. When a process inserts a new referring tuple, it must possess and use appropriate authority to vouch for the implicit information that may be leaked by the insert, as well as the information that might be revealed by any subsequent attempt to remove the referenced tuple. The following rule is enforced:

Rule 5.1. (Foreign Keys – Secrecy)
To insert a tuple a with secrecy label $a.L_S$, which is constrained to refer to a tuple b with secrecy label $b.L_S$, the process issuing the command must declassify for each tag in the symmetric difference of the two tuples' secrecy labels, $a.L_S \ominus b.L_S$.

The symmetric difference of two labels L_1 and L_2 consists of the union of $L_1 \setminus L_2$ (all the tags in L_1 but not L_2) and $L_2 \setminus L_1$ (all the tags in L_2 but not L_1). As stated above, the rule only handles secrecy; integrity is discussed in section 5.3.3. The justification for the rule comes in two parts:

— The requirement that the process must declassify for $b.L_S \setminus a.L_S$ addresses the problem with inserts described in the preceding section. In effect, the process must be able to see tuple b (if it exists) in order to verify the constraint, as if by the following pseudocode:

```
procedure insertReferencing(a, b):
  addSecrecy(b.L_S \ a.L_S)
  read b; abort if no tuple b is visible
  declassify(b.L_S \ a.L_S)
  insert(a)
```

The process effectively becomes contaminated because it must verify that b exists before inserting a. Thus, it needs to declassify in order to verify the constraint, and the rule merely makes this operation explicit.

— The requirement that the process must declassify for $a.L_S \setminus b.L_S$ addresses the problem with deletes described in the preceding section. The rule recognizes that deletions expose about information in the referencing table; it ensures that this is acceptable by requiring the inserter to have authority for all the tags that must be removed in order for the deleter to learn about these insertions. More concretely, the insert of a means that a subsequent attempt to delete b might fail and thus leak the existence of a to a process running with secrecy label $b.L_S$. By declassifying the tags that are in $a.L_S$ but not in $b.L_S$, the process performing the insert vouches that this potential unsafe flow is acceptable.

The affected tags must be identified explicitly in the INSERT statement using a DECLASSIFYING clause. Figure 5-2 shows two foreign key relationships where DECLASSIFYING clauses can be used. In figure 5-2(a), public information references

ActiveOrderStatus

_label	order_id	ready
{}	983462	true
{}	983463	false

Foreign key (order_id) references PharmacyOrders(id)

PharmacyOrders

_label	id	patient_name	patient_dob	medication
{bob_medical}	983462	Bob	6/26/1978	albuterol
{alice_medical}	983463	Alice	2/13/1960	warfarin
{cathy_medical}	983464	Cathy	4/22/1941	capecitabine

Primary key (id)

(a) Public information about the status of active pharmacy orders references the sensitive order details.

PatientVisits

_label	patient_name	patient_dob	visit_date	proc_code
{cathy_medical}	Cathy	4/22/1941	1/14/2012	J1700
{alice_medical}	Alice	2/13/1960	6/15/2012	G0404
{bob_medical}	Bob	6/26/1978	6/16/2012	J1700

Foreign key (proc_code) references ProcedureCodes(code)

ProcedureCodes

_label	code	cost	description
{}	G0404	50	electrocardiogram
{}	J1700	35	hydrocortisone injection
{}	Q0084	120	chemotherapy by infusion
{}	V2624	30	polishing artificial eye

Primary key (proc_code)

Source: *Healthcare Common Procedure Coding System*, Level II, 2012

(b) Sensitive patient visit records refer to public medical procedure codes.

Figure 5-2: The tables demonstrate two foreign key relationships that don't reveal significant information. In IFDB, applications use DECLASSIFYING clauses to vouch that these relationships are okay. Integrity tags are omitted to simplify the example.

sensitive information. The pharmacy maintains a table of active pharmacy orders that indicates which orders are ready. The entries in the table refer to sensitive information about the orders themselves via a foreign key. Inserting a new entry into ActiveOrderStatus doesn't reveal anything other than the existence of a particular order number, so the inserting process uses a DECLASSIFYING clause as follows:

```
INSERT INTO ActiveOrderStatus (order_id, ready)
    VALUES (983464, false)
    DECLASSIFYING (cathy_medical);
```

Figure 5-2(b) illustrates a case where sensitive tuples refer to public tuples. A hospital uses the PatientVisits table to keep track of what medical procedures are performed on patients. Medical information systems use standardized procedure codes for billing purposes; information about the cost of each procedure is stored in a separate ProcedureCodes table. A foreign key constraint enforces the requirement that each procedure code in the PatientVisits table must refer to a valid code in the ProcedureCodes table. An attacker who can delete procedure codes can use the constraint to determine what medical procedures have been performed at the hospital. However, in a large clinic, this isn't a serious threat to patient privacy, and furthermore the ProcedureCodes table can simply be protected from modification through the use of access control or integrity tags. Thus, a process inserting a new patient visit for Cathy uses a DECLASSIFYING clause as follows:

```
INSERT INTO PatientVisits (patient_name, patient_dob, visit_date, proc_code)
    VALUES ('Cathy', '4/22/1941', '6/30/2012', 'Q0084')
    DECLASSIFYING (cathy_medical);
```

DECLASSIFYING clauses are explicit declarations that any information flow channels created due to an insert of a tuple with a foreign key reference are acceptable. The process performing the insert must additionally have authority to declassify all the tags named in the clause. Therefore, the foreign key rule supports IFDB's goal that unsafe information flows can only occur through explicit use of authority.

Although section 5.3.1 demonstrates that some foreign key constraints have the potential to leak significant information, the tables in figure 5-2 show that some foreign key constraints don't leak any information that would be useful to an attacker. Thus, as a matter of convenience, it might be desirable to assert that the constraint is "harmless," so that processes don't require authority to insert. In IFDB, this can be done by creating a stored authority closure to do the insert and writing a rewrite rule that transforms insert statements into calls to the closure. IFDB does not provide

a more direct way to express this because it is not completely safe: it violates the requirement that only processes running with authority for a tag should be able to leak information protected by that tag. Specifically, a pair of collaborating malicious processes with no authority could abuse the constraint and the authority of the closure to leak arbitrary information at a high rate. For instance, one process becomes contaminated by the alice_password tag and inserts a pharmacy order with id *n* if the *n*th bit of Alice's password is a 1. The other process runs with an empty secrecy label and inserts an entry for order *n* into OrderStatus; it learns the *n*th bit of the password based on whether this insert succeeds or not. Nevertheless, "harmless" constraints might be acceptable in deployments where the goal is merely to protect against bugs, not malicious code.

Updates can be regarded as a delete followed by an insert, and thus, update statements also require DECLASSIFYING clauses. However, these clauses are only required when the update changes a field that is part of a foreign key reference into another table. Changes to other fields don't affect the constraint.

5.3.3 Referential Constraints and Integrity Labels

The problems with referential constraints and secrecy have analogues for integrity, although the practical consequences of integrity "leaks" are less serious. The problems are as follows:

— *Deletes.* If low-integrity processes can create references to high-integrity tuples, then those processes can interfere with the ability of high-integrity processes to delete those tuples. For example, suppose that appointment records in a medical clinic refer to doctor tuples via a foreign key. If Bob is fired from his job as a doctor at the clinic, he could prevent a high-integrity process from removing his doctor tuple by creating bogus appointments.

— *Inserts.* A process with high integrity that inserts a tuple referring to a low-integrity tuple is exposing itself to possible errors, because the success of the operation depends on the low-integrity tuple.

The problems can be addressed by extending rule 5.1 to handle both secrecy and integrity. As with secrecy, the goal is not to prevent the channel, which exists because the constraint is enforced between tuples with different labels. Rather, the

goal is to ensure that when unsafe flows do arise, the programmer acknowledges and properly vouches for them.

Rule 5.2. (Foreign Keys)
To insert a tuple a labeled $(a.L_S, a.L_I)$, which is constrained to refer to a tuple b labeled $(b.L_S, b.L_I)$, the process issuing the command must declassify for each tag in the symmetric difference of the two tuples' secrecy labels, $a.L_S \ominus b.L_S$, and it must endorse for every tag in the symmetric difference of their integrity labels, $a.L_I \ominus b.L_I$.

Endorsing for $a.L_I \setminus b.L_I$ is necessary because the process running with integrity $a.L_I$ needs to read the lower-integrity tuple b in order to verify the constraint; thus, a lower-integrity process could interfere with the high-integrity process by deleting b. Endorsing for $b.L_I \setminus a.L_I$ handles the case where a low-integrity tuple references a high-integrity tuple; the process performing the insert must vouch for the fact that it is interfering with the ability of higher-integrity processes to delete the tuple.

Rule 5.2 demonstrates that this thesis handles secrecy and integrity uniformly. In both cases, it uses a strong definition, *noninterference* [57], as a basis for reasoning about covert channels. In essence, noninterference requires that low-secrecy parts of the system should not be affected by high-secrecy parts of the system (lest they learn about high-secrecy data), and high-integrity parts of the system should not be affected by low-integrity parts of the system. Thus, secrecy and integrity are considered duals of one another.

Treating secrecy and integrity as duals leads to a clean design that is easy to explain, because there is only a single set of rules, which works for both kinds of labels. However, from a practical perspective, forcing programmers to add DECLASSIFYING and ENDORSING clauses to their SQL statements and run code with additional authority comes at a cost. In the case of integrity, it is not clear that this cost is justifiable. In essence, the problems prevented by rule 5.2 are denial-of-service attacks. One might be perfectly happy with an integrity rule that allows such attacks, but prevents problems such as trusted medical applications inadvertently using bogus medical records that don't have the proper integrity. Thus, for many applications, integrity semantics based on less strict notions than noninterference may be more appropriate [93, 149], and future work might introduce alternative integrity rules for foreign key constraints in a database. Section 11.2.2 discusses the issue further.

5.4 General Constraints

In addition to the constraints discussed so far, there are other types of constraints that aren't expressible in standard SQL, except via assertions or triggers. Some examples include the following:

— *Equational dependencies* define algebraic constraints, possibly over aggregate values. For example, the sum of all loan amounts may not exceed the bank's credit limit.

— *Inclusion dependencies* are similar to foreign keys, except that the referenced tuple need not be unique. For example, each prescribed drug must be listed *at least once* in the pharmacy catalog – but the catalog may have multiple listings, each for a different manufacturer or formulation.

— *Exclusion constraints* generalize uniqueness constraints to support requirements such as, "No two salespeople should be within 100 miles of each other."

Like most database systems, IFDB doesn't provide any special support for such constraints. Instead, constraints other than those covered in preceding sections of this chapter must be enforced by triggers. Triggers in IFDB are simply stored procedures that are executed in response to specified types of writes (inserts, updates, or deletes) on a particular table. Triggers may enforce a constraint by aborting; by modifying the requested operation; or by performing an additional operation, such as updating a bank balance every time a new deposit or withdrawal tuple is inserted.

It is important to understand the information flow implications of constraints implemented through triggers. This chapter has developed semantics for two useful kinds of constraints: uniqueness constraints and foreign keys. Many triggers are similar to these, and therefore, similar rules may be appropriate. For example, inclusion dependencies are similar to foreign keys, while exclusion constraints resemble uniqueness constraints. However, there is no general theory about how to handle the information flows that might arise due to relationships among tuples governed by arbitrary constraints. Appropriate semantics must be motivated by the type of constraint and the needs of the application.

Nevertheless, there is a methodology for writing triggers. The database designer has two choices. One choice is to make the trigger an ordinary stored procedure, in which case it runs with the authority of the process attempting to modify the table.

In this case, the trigger can only enforce the constraint with respect to the subset of the database visible to the process. However, enforcing the constraint will not result in any unsafe flows because the trigger does not do anything that the caller could not do itself. The second choice is to make the trigger a stored authority closure, in which case it can use its authority to enforce the constraint across all tags that it has authority for. However, by using authority, the trigger could create unsafe flows and expose information. Therefore, as with any other authority closure, it is important to consider how that authority might be abused.

Figure 5-3 contains the definition of a checkMedicationExists trigger that demonstrates the first choice. The trigger procedure does not use any authority, so it cannot leak any information. It is attached to the PharmacyOrders table in figure 5-2(a), and it implements half of a foreign key constraint: the procedure checks on insert into PharmacyOrders that the order refers to a valid medication, which must be visible to the process. However, it does not check whether writes to Medications (not shown) invalidate existing pharmacy orders. Triggers like this one are useful: since the trigger implements a constraint that is weaker than a full foreign key constraint, the inserting process does not need the same authority as it would if a foreign key constraint had been used (see section 5.3). Furthermore, this weaker constraint is reasonable if updates to the Medications table are restricted to trusted processes via integrity tags or an access control policy.

The trigger shown in figure 5-4 is an example of the second choice – running a trigger with authority. The trigger enforces the constraint that all tuples in the UserAccounts table have unique values for the username field. If each account has a different label, for instance {alice_account} for Alice's account, this is an unsafe flow: attempting to insert a new account tuple might leak the fact that a certain username already exists. Normally IFDB prevents unsafe flows like this one by using polyinstantiation (section 5.2), but leaking the existence of a username when another user tries to register the same username is both benign and unavoidable. By declassifying inside the closure, the programmer acknowledges that this leak is acceptable. It is easy to reason about what is being leaked: The only action the closure takes is to succeed or fail based on whether there is an existing account with the given username. Therefore the procedure can only leak that one bit of information by declassifying.

Triggers (as well as other types of constraints) can be evaluated at the time of the action that caused the trigger to fire, or if the action is part of a larger transaction, they can be deferred until the entire transaction commits. (Chapter 6 covers transactions

```
CREATE FUNCTION checkMedicationExists()
RETURNS trigger AS
$$
BEGIN
        PERFORM * FROM Medications WHERE medication = NEW.medication;
        IF NOT FOUND THEN
                RAISE EXCEPTION 'medication % does not exist', NEW.medication;
        END IF;
        RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER prescriptionMedicationTrigger
        BEFORE INSERT ON PharmacyOrders
        FOR EACH ROW EXECUTE PROCEDURE checkMedicationExists();
```

Figure 5-3: The checkMedicationExists procedure runs as a trigger and ensures that medications named in new pharmacy orders actually exist in the Medications table. The procedure does not use any authority, so it cannot cause any information leaks.

```
CREATE FUNCTION checkAccountExists()
RETURNS trigger PRINCIPAL accountAdmin AS
$$
DECLARE
        existingAccount VARCHAR;
BEGIN
        PERFORM addSecrecy(all_user_accounts);
        SELECT * INTO existingAccount FROM UserAccounts
                WHERE username = NEW.username;
        PERFORM declassify(all_user_accounts);
        IF existingAccount IS NOT NULL THEN
                RAISE EXCEPTION 'user % already exists', NEW.username;
        END IF;
        RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER accountUniqueTrigger
        BEFORE INSERT ON UserAccounts
        FOR EACH ROW EXECUTE PROCEDURE checkAccountExists();
```

Figure 5-4: The checkAccountExists stored authority closure is invoked as a trigger on each insert into the UserAccounts table. It runs with authority for the all_user_accounts compound tag, and it uses its authority to ensure that all user accounts that are properly labeled with subtags of all_user_accounts have unique usernames. Note that although the trigger declassifies the all_user_accounts, the semantics for authority closures ensures that it cannot inadvertently remove any account tags from its caller's label (see section 4.4).

in IFDB.) The former option catches errors sooner, but the latter is necessary in certain situations. For example, if a constraint specifies a one-to-one mapping between tuples in two tables, checking the constraint after each insert wouldn't work; the check must be deferred until after both tuples are inserted. Deferred triggers require some extra bookkeeping in IFDB: they run with the information flow labels that the process had at the time of the action that caused the trigger to fire, rather than the labels of the process at the time the transaction commits. This ensures that triggers have consistent semantics regardless of whether they are deferred until the end of the transaction. For example, if a process inserts a tuple and then declassifies, a trigger that is validating the insert should see the subset of the database that was visible to the process before it declassified.

5.5 Constraints on Labels

Label constraints, also known as classification constraints, restrict the labels that tuples are allowed to have. The SeaView multi-level secure DBMS [101] first introduced the idea as a way "to protect against labeling errors as well as to relieve the user of the burden of remembering all of the rules for classifying data" [37]. IFDB also supports label constraints, but has a different philosophy about these constraints. In IFDB, the supposition is that most database interaction happens through applications, not through human users. Furthermore, these applications are subject to information flow restrictions as well. (In contrast, human operators are less amenable to information flow restrictions.) Therefore, label constraints are used differently, and can in fact be used to eliminate some of the issues identified earlier in this chapter, such as polyinstantiation. This section lists some of those uses:

— *Preventing labeling errors.* As in SeaView, label constraints can prevent data from being labeled incorrectly. Unlike SeaView, IFDB uses the DIFC model, so such constraints usually require referring to another table to determine the appropriate label. For example, figure 5-5 shows how to use foreign keys to enforce the constraint that patient visit records have the same secrecy label as the respective patient's basic medical information. More complex constraints, such as one that ensures that a tuple must have *at least* a certain tag in its secrecy label, can be enforced with triggers. Triggers have two advantages: they are more general, and they avoid some of the issues with foreign keys discussed in section 5.3.

PatientVisits

_label	patient_name	patient_dob	visit_date	proc_code
{cathy_medical}	Cathy	4/22/1941	1/14/2012	J1700
{alice_medical}	Alice	2/13/1960	6/15/2012	G0404
{bob_medical}	Bob	6/26/1978	6/16/2012	J1700

Foreign key (_label, patient_name, patient_dob) references Patients

Patients

_label	patient_name	patient_dob	blood_type	allergies
{alice_medical}	Alice	2/13/1960	O+	
{bob_medical}	Bob	6/26/1978	A-	bees
{cathy_medical}	Cathy	4/22/1941	B+	

Primary key (_label, patient_name, patient_dob)

Figure 5-5: A foreign key relationship between the PatientVisits table and the Patients table ensures that patient records are consistently labeled.

A particularly useful constraint that can be enforced with a trigger is that all patient medical records should be labeled only with tags that are subtags of the compound tag all_patients_medical. Such a constraint forces all the tags to be of the right "type," and ensures that no additional contamination is present in the labels. With this constraint, it is possible to run a statistical computation with label {all_patients_medical} that is guaranteed to observe all patient records.

— *Preventing polyinstantiation.* Another invariant that should be enforced on the tables in figure 5-5 is the constraint that there is only one Patients record for Alice. Polyinstantiation (section 5.2) creates the possibility that two records might exist with different labels.[2] It is possible to prevent polyinstantiation by requiring that each patient tuple have the *correct* label. The mapping between patients and their tags could be stored in another table, say PatientTags, and checked by a trigger on each insert into Patients. This solution would seem

2. SQL requires that the target of a foreign key reference be covered by a uniqueness constraint. Therefore, the secrecy label has actually been included in the primary key of Patients. Doing this technically prevents polyinstantiation, but on a semantic level, it is still wrong to have multiple Patients tuples for the same patient.

to lead to an infinite regress: how does one prevent polyinstantiation in the PatientTags table? One answer is to constrain all tuples in the table to have empty secrecy labels. If the table were indexed by an opaque identifier such as medical record number, this would be a fine solution. Such a table is commonly needed anyway, so that processes know what tags to add to their labels to read a particular patient's records. An alternative is to protect the contents of the PatientTags table with a tag associated with an authority closure that uses the information only to verify that patient records are properly labeled.

— *Preventing applications from seeing dangling references.* Although IFDB enforces referential integrity constraints, applications may observe apparent violations of these constraints because they don't have the proper labels to see the referenced tuple. In some cases, this is deserved. For example, if there is a constraint that every user has a password, that does not mean that a process that sees Bob's contact information ought to be able to read his password without first raising its label. However, if a label constraint requires the referenced tuple to have the "correct" labels, then any process with those labels will observe no violations of referential integrity. Another example is the one in figure 5-5: no process will ever observe a violation of the foreign key constraint, because the referring and referenced tuples are constrained to have identical labels.

This chapter has argued that there is a fundamental tension between constraint enforcement and information flow control: sacrifices in integrity constraint enforcement must be made to avoid covert channels. Supporting examples have shown that badly behaved processes can abuse constraints to learn sensitive information protected by those constraints. The insight of this section is that incorporating appropriate label constraints to restrict bad behavior can guarantee both data consistency and freedom from covert channels.

5.6 Summary of Contributions

This section briefly summarizes the contributions of this chapter and highlights how IFDB differs from prior work. Section 10.4 presents the related work on constraints in multi-level-secure databases in greater detail.

The ideas of polyinstantiation and label constraints were introduced in SeaView [101]; IFDB's contributions here are simplifying the semantics of polyinstantiation and advocating a practical methodology for managing polyinstantiation. In particular, the use of label constraints to prevent polyinstantiation safely is new.

Additionally, IFDB pioneers the idea of using DECLASSIFYING and ENDORSING clauses to vouch for the information flows created by foreign key constraints. Prior systems either allowed unsafe flows or disallowed foreign key constraints between tuples with certain labels. Finally, this chapter develops new semantics and a methodology for using triggers to enforce general of constraints.

Chapter 6 Transactions

This chapter discusses covert channels that can arise due to support for transactions, and describes how IFDB eliminates the channels. Transactions can create two types of channels. The first has to do with the fact that processes can leak information by choosing to abort transactions, while the second is related to conflicts among transactions issued by processes with different labels.

An important assumption made in this chapter is that dirty reads are not allowed; transactions may not read uncommitted data from other active transactions. Dirty reads create the possibility of cascading aborts and unrecoverable schedules [61], and they complicate the reasoning about information flows. IFDB is based on PostgreSQL, which does not allow dirty reads. The problems and solutions presented in this chapter are not specific to a particular isolation level, however; they apply to serializability, snapshot isolation, SQL's `REPEATABLE READ`, and so forth.

6.1 Label Changes and Aborts

It is crucial that processes be able to write tuples with different labels as part of a single transaction. For example, when a new user signs up for an account on a website, the user's contact information tuple may have a different secrecy label from her password tuple, but both tuples should be added to the database as a single atomic action. Section 4.3.5 explains that IFDB makes this possible by allowing processes to change their labels in the middle of a transaction.

A client process can abort a transaction after the process has raised its labels, however, and this can lead to a covert channel. Specifically, the decision of whether to commit or abort can leak one bit of information per transaction. The following code listing, written in PL/pgSQL, shows how a process might abuse the channel to leak some of Alice's medical information to a collaborator who has an empty secrecy label. The example makes use of the HIVPatients table from figure 5-1 on page 64.

```
BEGIN
    INSERT INTO Foo VALUES ('Alice has HIV');
    PERFORM addSecrecy(alice_medical);
    SELECT * FROM HIVPatients  WHERE pname='Alice';
    IF NOT FOUND THEN
        ABORT;
    END IF;
COMMIT;
```

This code begins a transaction and writes the string "Alice has HIV" with an empty secrecy label. Then it raises the secrecy label of the process and checks whether Alice actually has HIV. The process is now contaminated, so it is unable to release the sensitive information it read directly, assuming it has no authority to declassify the alice_medical tag. However, the process reveals the information indirectly by committing the transaction if Alice has HIV and aborting otherwise. The result is that the string "Alice has HIV" is written with an empty secrecy label if and only if Alice has HIV. Subsequently, another process with an empty secrecy label could read the string from table Foo and release this sensitive information to the world. Although the example contains malicious code, the fact that aborts can occur due to errors raises the prospect that such leaks could also occur through exploitation of buggy code. In fact, inducing errors to reveal sensitive information is a well-known attack [110].

One could imagine trying to solve the problem by restricting the circumstances under which aborts are allowed. Fabric [96] does this, as explained in section 10.6.1. However, clients can cause transactions to abort in many ways, such as by issuing invalid operations, attempting to violate integrity constraints, or inducing deadlocks; preventing all of these in a dynamic system with general transactions is unrealistic.

The intuition for fixing the problem is that the information flow rules should apply at the commit point, which is when the writes become visible to other transactions. (Dirty reads would invalidate this line of reasoning, but they are not allowed.)

IFDB enforces the rule that the *commit label*, that is, the process label at the commit point, must be no more restrictive than any tuple in the transaction's write set. If a process violates the rule by attempting to commit a transaction with a higher label than allowed, the result is an error, which causes the transaction to abort. Formally, the rule is as follows:

Rule 6.1. **(Transaction Commit)**
Let $\mathcal{W}_T = \{\tau_1, \ldots, \tau_n\}$ denote the set of tuples written by a transaction T issued by process P. P can commit T if $\forall \tau_i \in \mathcal{W}_T$: $(P.L_S, P.L_I) \preceq (\tau_i.L_S, \tau_i.L_I)$, where $(P.L_S, P.L_I)$ is the current label pair of P.

The rule effectively means that transactions running without authority cannot commit if they become more contaminated after performing a write. In particular, the transaction in the example cannot leak whether Alice has HIV, because it will always abort. However, a process that has authority for a tag can add that tag to its label and subsequently remove it before the end of the transaction in order to write differently labeled tuples atomically. Such a transaction could expose information, but this is allowed because it used the proper authority to declassify the tag in question.

Figure 6-1 provides some intuition for why the rule is necessary and sufficient by showing three transactions that satisfy the rule and three that do not. Most transactions in practice resemble T_1: no label changes occur during the course of the transaction. Transactions are also allowed to raise their labels prior to performing any writes, as T_2 does. T_2 is safe because the same sequence of reads and writes could have been done by T_1. T_2 has more restrictive labels than T_1 for the first read, but by rule 4.1, T_1 it can read anything that T_2 can read.

T_3 writes tuples with different labels and is also permitted by the rule. This transaction roughly captures the motivating example given at the beginning of this section: adding a new user account involves writing a contact tuple with one label pair and a password tuple with a different label pair. The process that creates accounts is trusted to protect the account information it is adding; it declassifies the password tag to vouch that nothing it did while contaminated with that tag will affect the outcome of the transaction in a dangerous way. (Of course, the contact and password information may have incomparable labels, which can't be represented in the figure; for instance, the secrecy labels might be {alice_contact} and {alice_password}. In this case, the process must have authority for both tags so that it can reduce its secrecy label to the greatest lower bound of the two labels, which is {}.)

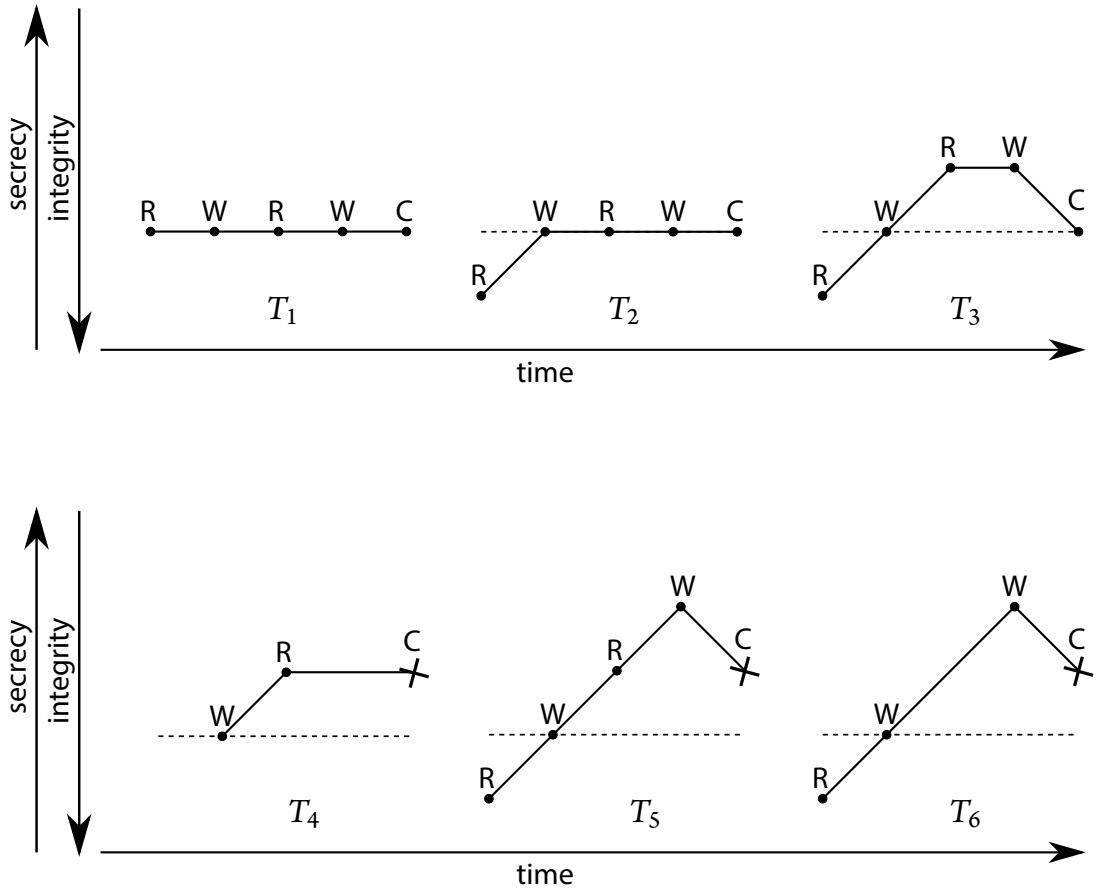


Figure 6-1: The timeline shows the labels of six transactions as they perform a series of reads and writes, and finally attempt to commit. For each transaction, the dotted line shows the maximum label that the transaction would be allowed to commit with.

T_4 has exactly the same form as the HIVPatients example, and T_5 is just T_4 with some additional operations. Rule 6.1 does not allow either transaction to commit, thus preventing them from introducing unsafe flows. The rule also disallows T_6, but this raises an interesting question: is the rule too strict? T_6 is like T_5, but it does not perform any reads of higher-labeled tuples. Thus, it would seem that T_6 is not subject to the problem demonstrated by the HIVPatients example. However, the write T_6 performs after it raises its label can still influence its behavior via integrity constraint violations and concurrency conflicts. Therefore, committing T_6 would still potentially lead to a covert channel. The rule prevents this problem by forcing T_6 to abort regardless.

Concurrency conflicts aren't a concern just for T_6. For instance, suppose T_2's first write affects the same tuple as T_4's first write. If T_4 were to decide whether to stall or abort on account of the secret it subsequently reads, this might introduce an unsafe flow if it affects the outcome of T_2. This section has addressed information flow issues only for independent transactions; the next section explains how IFDB prevents concurrency conflicts among transactions from introducing covert channels.

6.2 Conflict Channels

Concurrent transactions pose a potential problem because conflicts that involve transactions with different labels could introduce signaling channels. Without adequate precautions, a transaction with low secrecy might be blocked or aborted due to the actions of a high-secrecy transaction, or a high-integrity transaction might similarly be affected by a low-integrity transaction. Clients can keep transactions open for unbounded amounts of time, so such channels are easy to exploit.

Additionally, contention for shared resources such as memory, disk I/O bandwidth, and CPU can introduce timing channels, causing transactions to block temporarily due to the activity from other clients. Such channels can be mitigated, but this is an implementation concern. This section considers only covert channels in the model; in effect it assumes an infinitely fast database. For example, if a high-integrity transaction cannot commit because its writes conflict with those of a low-integrity transaction, that is a problem in the model. If the high-integrity transaction is merely delayed for a bounded amount of time due to I/O contention with the low-integrity transaction, that is a problem in the implementation. Timing channels in the implementation are discussed in section 8.5.1.

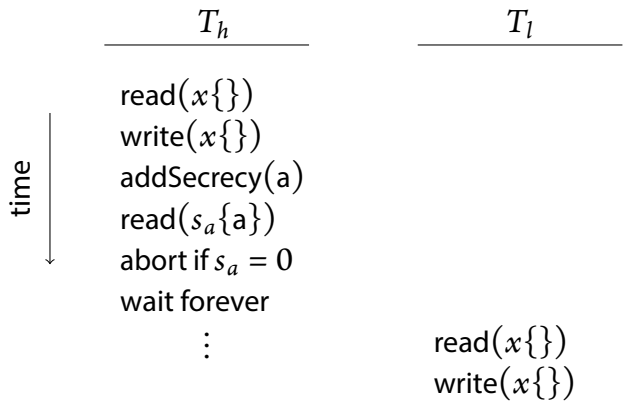


Figure 6-2: A conflict between a high-labeled transaction T_h and a low-labeled transaction T_l could potentially introduce a covert channel. T_h and T_l both start with empty labels, and then T_h raises its label. The notation write(x{}) means that the transaction writes tuple x, which has label {}.

Figure 6-2 illustrates a conflict channel. Two processes P_h and P_l start with empty labels, and P_h starts a transaction T_h. T_h updates tuple x, then raises its secrecy label to {a} and reads a secret s_a. Since T_h wrote a tuple with an empty secrecy label, rule 6.1 will not allow it to commit with secrecy label {a}. However, depending on whether s_a is zero, P_h either aborts T_h or stalls indefinitely. Subsequently, P_l starts transaction T_l, which also attempts to update x. Thus, T_l conflicts with T_h, but only if s_a is nonzero. If the DBMS were to abort T_l or force T_l to block until T_h finishes, P_l could learn whether the secret s_a is zero. This flow is unsafe, since neither P_h nor P_l required authority for the tag a. There are two ways to address the problem:

— *Transaction Clearance.* After a transaction performs a write, the clearance of the process that issued it is limited until the end of the transaction. Specifically, the process cannot add secrecy tags or remove integrity tags that it does not have authority for. This restriction is reasonable because rule 6.1 states that a transaction is doomed to abort if its label when it tries to commit is more restrictive than any of the tuples it wrote.[1] In the example, this rule prevents P_h from adding secrecy tag a unless it has authority for that tag.

1. A few kinds of computations are allowed by rule 6.1 but disallowed with transaction clearance. An example is a process that starts a transaction and adds secrecy tags in a reduced authority call, leaving it to the higher-authority caller to remove those tags.

— *Flow-Safe Scheduling.* The transaction manager can schedule transactions and handle transaction conflicts in such a way that conflicts never result in unsafe flows. For instance, it is safe to abort T_h in the preceding example, since P_h's label is more restrictive than P_l's.

Flow-safe scheduling provides stronger guarantees than transaction clearance. Specifically, flow-safe scheduling meets the goal that all unsafe flows should be explicit; it ensures that conflict channels can only introduce unsafe flows of information protected by a given tag if a process explicitly declassifies or endorses for that tag in the middle of a transaction. In contrast, transaction clearance allows transactions to introduce unsafe flows if the process that issued them merely had authority for the tag, even if it did not use that authority yet.

However, the transaction clearance approach is simple and ultimately more practical. Flow-safe schedulers introduce performance penalties, either by severely limiting concurrency or creating the potential for starvation. Furthermore, conflict channels are rare, and it's hard to imagine trusted code inadvertently leaking significant information via such channels. Appendix A describes how to do flow-safe scheduling in IFDB, and explains the drawbacks involved.

Chapter 7 Case Studies

In evaluating IFDB, there are three important questions to answer: First, does IFDB provide abstractions that make DIFC easy to use in real applications? Second, how well does IFDB improve information security? Third, how well does it perform? This chapter addresses the first two questions, while chapter 9 addresses the last question.

Two applications, CarTel and HotCRP, were ported to use IFDB and PHP-IF. These applications are good candidates because they have rich policies for sharing information among users. Furthermore, both of them store data in relational databases, and both are written in PHP. The conversion effort addressed only confidentiality concerns, not integrity. Although IFDB was designed as a way to build security into *new* database-backed applications, the conversion of these existing applications required only a modest effort: 4.5% of the CarTel code base and 7% of the HotCRP code base were changed. In the latter case, most changes are related to the fact that HotCRP was designed for MySQL and made numerous nonstandard assumptions. Therefore, it had to be modified to work with IFDB, which is based on PostgreSQL.

Like most web applications, the original versions of CarTel and HotCRP put complete trust in all of the application code. Therefore, bugs in any part of the code had the potential to compromise privacy. In fact, both applications were vulnerable to several leaks, and converting them to use IFDB fixed these leaks. One previously unknown bug in HotCRP leaked the contact information for all the conference participants, and IFDB prevented the leak. Additionally, two leaks from older versions of HotCRP were reintroduced, and IFDB prevented those leaks as

well. Using IFDB also fixed three types of vulnerabilities in CarTel. Additionally, much less application code had to be trusted when using IFDB, which made security easier to reason about.

7.1 CarTel

CarTel [70] is a mobile sensor network that collects location data and other information from GPS-equipped cars. Users can see maps and statistics about their past drives through the CarTel website, get real-time traffic information derived from other users' drives, and compare their drives with their friends' drives.

7.1.1 The CarTel Implementation

The present CarTel implementation is a prototype produced by a research project studying mobile sensor networks. Vehicle location readings are sent to a central portal called the *track server*, where they are stored in a relational database. The track server is written in Python.

Two interfaces are available to extract data. A streaming database, ICEDB, supports continuous queries such as real-time traffic information for mobile applications. A web portal provides users with information about past drives, as well as current car locations. The focus here is on the web portal, which uses *ad hoc* privacy controls enforced by PHP scripts running on the web server. There are sixty-two such scripts. Each PHP script has complete access to all users' location data, and is trusted to ensure that data aren't released inappropriately. This is a common design, but it led to many security bugs.

Figure 7-1 shows the relevant parts of the schema in CarTel before it was ported to use IFDB. All user account information, including usernames, contact information, and passwords, is stored in the Users table. The Friends table stores settings related to how users have chosen to share their data with others; the concept is discussed in the following section. The Cars table stores records about each user's cars, including the ID and the user-assigned name of the car.

The track server inserts raw location data into the Locations table. Triggers and PHP code process the location readings and break them into *drives*, which are stored in the Drives table. The system considers a car to have started a new drive if the car has not moved significantly over a five-minute period. To speed up common queries,

Table	Contents
Users	usernames, contact information, and passwords
Friends	friender-friendee mappings and settings
Cars	information about users' cars
Locations	current and historical car locations
LocationsLatest	latest location of each car
Drives	past drives
LastDrive	most recent drive for each car

Figure 7-1: The original CarTel schema contains tables for user account data, friend relationships, location data, and information about users' cars. Raw location readings are inserted into Locations initially, and subsequently processed into discrete drives.

the LocationsLatest and LastDrive tables track the most recent location and drive for each car. They are kept up-to-date by triggers in the Locations and Drives tables, respectively.

CarTel uses primary keys (and hence uniqueness constraints) in the larger tables, such as the ones that store location data. There are no DBMS-enforced foreign key constraints, but there are a number of implicit constraints; for instance, each location and each drive refers to a valid car. Additionally, the relationships between some pairs of tables, such as Locations and CurrentLocation, are maintained by triggers as discussed previously.

7.1.2 Security Requirements

CarTel is intended to protect users' privacy: the records for a car should only be accessible to its owner, but the owner can allow friends to see certain information. Additionally, real-time traffic computations may use anonymized location data, but the focus here is on users and their friends.

There are six main types of user data to protect, each with a different security policy. There are also security policies for other types of information, such as map data and access logs, but these are not considered in this chapter. The relevant types of data are:

— *Names.* The usernames and full names of each user are available to all other users in order to allow users to designate which other users are their friends.

— *Contact Information.* Each user optionally provides an email address and street address, and this information is not shared with others.

— *Passwords.* A user's password should not be released, even to the user.

— *Current Locations.* A user can see the locations of his cars, and he can designate "friends" who are allowed to see this information as well.

— *Past Drives.* Similarly, a user can designate a (possibly different) group of "friends" who are allowed to compare their past drives with his.

— *Cars.* Each user and both types of that user's friends can see the names and icons associated with that user's cars.

To allow users to select friends, the list of usernames and their corresponding real names is public. Additionally, privacy policies for users' friend lists are not considered here; friend relationships are considered public knowledge.

7.1.3 Securing CarTel With DIFC

In the modified version of CarTel, IFDB and PHP-IF work together to enforce the policies stated in the preceding section. When a user Alice drives to work, the track server labels location readings from her car with tags she owns and inserts them into the database. Any web script that reads those entries becomes contaminated by those tags. Once contaminated, anything it writes back to the database is also contaminated, and it cannot communicate with the outside world, unless it has proper authority and declassifies the information.

Tags were introduced to handle the different types of confidential information. A single tag, sys:auth, protects all user passwords, and only the CreateUser, AuthenticateUser, and ChangePassword closures have authority for this tag. Each user, for instance Alice, has four tags for her data: alice_contact protects her address, alice_location covers her current location, alice_drives is for her past drives, and alice_cars is attached to information about her cars.

Labeling tuples properly required several changes. The Users table contained information governed by different security policies, so it was vertically partitioned into three tables: UserNames, UserContact, and Passwords. A Users view that joins the former two tables provides compatibility with existing code. No other schema changes were required, since the remaining tables were already partitioned according

Table	Secrecy Label	Contents
UserNames	{}	usernames and full names
UserContact	{*user*_contact}	user email and street addresses
Passwords	{sys:auth}	user passwords
User Tags	{}	secrecy tags for each user
Friends	{}	friender-friendee mappings
Cars	{*user*_cars}	information about users' cars
Locations	{*user*_drives, *user*_location}	current and historical car locations
LocationsLatest	{*user*_location}	latest location of each car
Drives	{*user*_drives}	past drives
LastDrive	{*user*_drives}	most recent drive for each car

Figure 7-2: CarTel user data are stored in IFDB tables, where each tuple has a user-specific secrecy label.

to confidentiality needs. Additionally, the track server must attach the appropriate tags before inserting the tuples into the database. By necessity, the track server is trusted with all users' data, and there was no attempt to reduce the amount of trust required in the track server. However, some changes were needed to ensure that location readings could be labeled efficiently, as discussed in section 9.2.2.

Figure 7-2 shows the modified schema and the labeling strategy. Tuples in the Locations table containing location readings for Alice's cars can be used to compute past drives, but they also reveal current locations if they are recent enough. Therefore, they are assigned the label {alice_drives, alice_location}. These tuples are subsequently processed to produce past drives, stored in the Drives table, and the latest locations of each of Alice's cars, stored in LocationsLatest. The past drives are labeled with {alice_drives}, and the latest locations are labeled {alice_location}.[1] Alice can allow her friend Bob to see one type of information or the other by delegating authority for alice_drives or alice_location to him; the appropriate delegations are added or revoked whenever Alice updates her friend settings.

A substantial amount of code in CarTel is involved in transforming raw location data into drives. Figure 7-3 shows the flows of location data. An SQL stored procedure, driveupdate, runs as a trigger and updates the distance traveled in the drive as new

1. The original CarTel implementation separated Locations from LocationsLatest for performance reasons. This separation happened to simplify the job of extending CarTel to use DIFC.

location readings come in. Separately, the PHP function load_drives interpolates the path of a drive. The latter computation is more expensive, so it is done on demand when Alice's drives are plotted on the website, and stored to speed up future requests. Even though driveupdate and load_drives process secret data, IFDB prevents them from compromising Alice's privacy. Both procedures read raw location data with label {alice_drives, alice_location} and write drives with label {alice_drives}. They run as authority closures with the ability to declassify for the alice_location tag; however, they cannot declassify for the alice_drives tag. Therefore, once the procedures read Alice's location information, they will not be able to release it to the outside world, and anything they write back to the database will be contaminated. At worst, they could expose Alice's current location to her friends who are only supposed to be allowed to see her past drives.

Some information flows in CarTel don't involve any declassification, and therefore they can occur without any use of authority. An example of such code is the updatelastdrive trigger, which runs after every insert into the Drives table and updates the LastDrive table. Similarly, none of the SQL and PHP code involved in computing distances and interpolating routes needs to be trusted. Most importantly, the application does not need to be trusted to produce the right queries, since Query by Label limits the results it sees.

The account creation, authentication, and session-management code needed to be largely rewritten to use closures that act for all users. Creating an account now requires making three tags and inserting four tuples (UserNames, UserContact, Passwords, and UserTags). In PHP-IF, web scripts run with no authority unless they first authenticate by calling the AuthenticateUser closure with the user's username and password. If the username and password are correct, the closure upgrades the caller's authority by invoking PHP-IF's switchTo primitive, introduced in section 8.4.2. Additionally, CarTel provides a trusted session-management closure that is invoked at the start of every script. CarTel session state does not contain any sensitive information; it only contains the authenticated user's username, principal, and tags. The session-management closure switches to the authenticated user's principal on every request.

7.1.4 Bugs Prevented

Section 7.1.3 explains how IFDB prevents the code that processes location data from exposing that data. Thus, using DIFC improved security in principle, by reducing

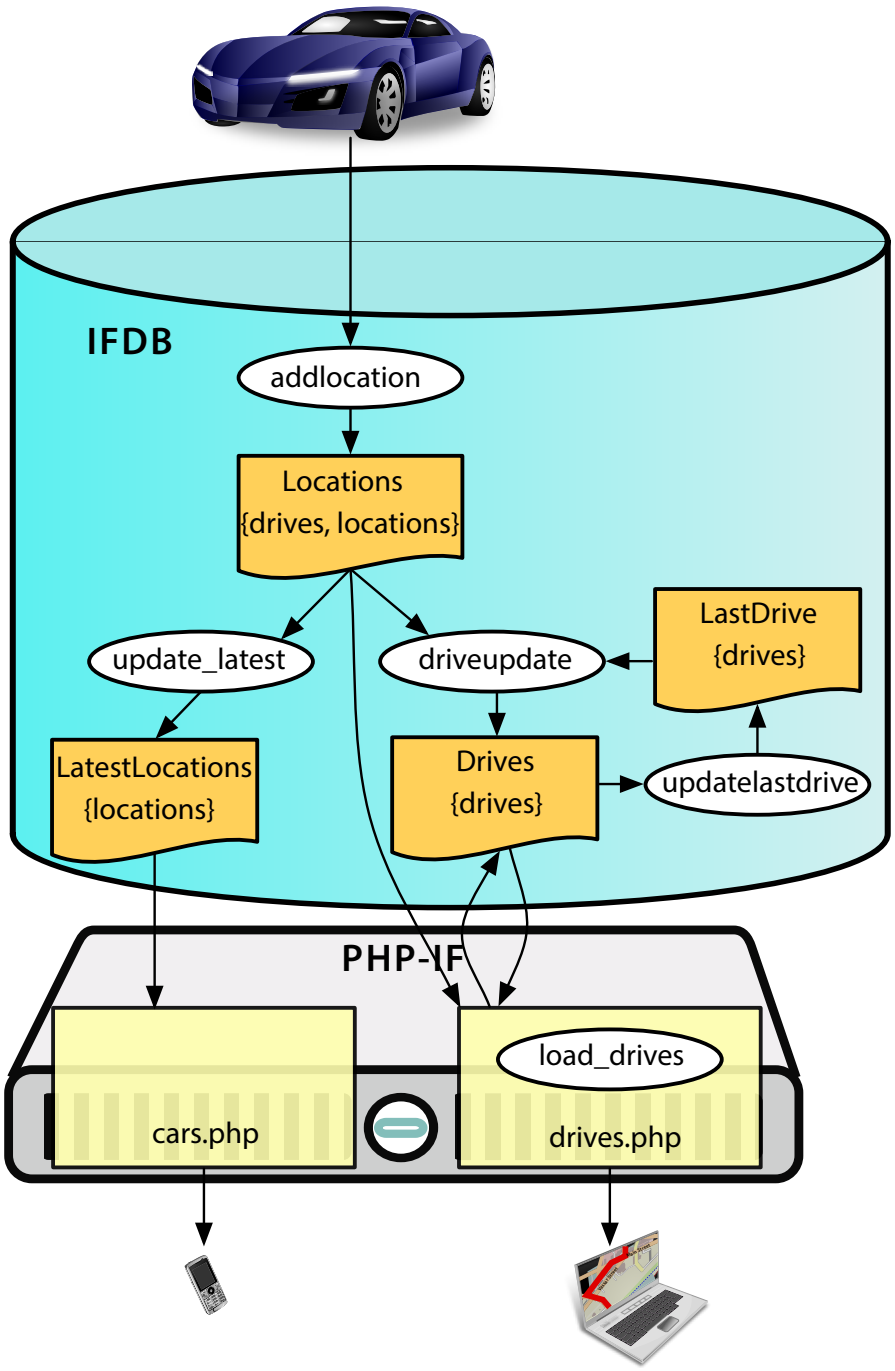


Figure 7-3: Location data in CarTel is processed by database stored procedures and PHP code. IFDB prevents bugs in this code from leaking information.

the amount of trusted code. IFDB also improved security in practice by preventing three security bugs from causing information leaks.

The first two types of bugs were in authentication. Twelve scripts, many of which were rarely used or intended only for testing, neglected to authenticate the user making the request. Second, the authentication routine itself also had a bug: when authentication failed, the script continued as if the user authenticated successfully, but it generated an HTTP redirect to send the user to the login page. The HTTP redirect hid the bug from honest users, whose browsers redirected them without displaying the rest of the page.

The third type of bug related to the "friend" feature. Users are supposed to be able to see the drives or current locations of their "friends" who have allowed such sharing. However, while some scripts included logic to select only cars of friends, three did not. Specifically, users were able to compare their drives with those of non-friends, see a calendar of the drives for any car given its ID, and see the current location of any car. The third instance of the bug involved old code that was not even normally used; however, the code could be invoked by a malicious user to produce the location of any car by passing in the HTTP query string carid=ID.

Converting CarTel to use IFDB fixed all of these bugs. Scripts that don't authenticate run with no authority, and therefore are unable to declassify sensitive information. If a user attempts to coerce the site into showing the drives for a non-friend, the script becomes contaminated with a tag it has no authority to declassify, and therefore it produces no output regardless of what it reads.

It's tempting to think that the kinds of bugs we found in CarTel are confined to research prototypes, but this is not the case. For example a bank and a health plan insurer, both Fortune 500 companies, recently exposed millions of their customers' financial and medical records due to omitted authentication checks [111, 130]. The missing authorization checks in CarTel mirror observed privacy holes in the Facebook website [30].

7.2 HotCRP

HotCRP [83] is a widely used conference management system. Authors submit papers, reviewers read them and enter evaluations, and the program committee (PC) produces decisions for the papers. The program chair orchestrates the process. The system is intended to handle conflicts of interest so that PC members cannot see

reviews or decisions for their own papers, or papers of their close colleagues, prior to decisions being released. A PC member is said to be *conflicted* with such papers.

7.2.1 The HotCRP Implementation

HotCRP is based on an earlier conference management system called CRP, and its design has evolved significantly since the original system was conceived. Many new features have been added over the years, often to meet various requirements and peculiarities of the conferences that have used the system. The implementation consists of about twenty-four user-visible scripts and twenty-four scripts containing library routines, for a total of about 25,000 lines of PHP code. It was designed to use the MySQL database as its backing store. The code depends on a number of non-standard behaviors of MySQL, so most database queries required modification to work with IFDB, which is based on PostgreSQL.

The security policy is enforced by the HotCRP scripts, which have access to all users' data. The privacy settings are highly configurable. The options available to the chair include the following:

— Paper authors can be anonymous or not.

— Review authors may be anonymous or not.

— PC members can be allowed access to non-conflicted paper reviews either before or after they have written their own assigned reviews.

— PC members can see acceptance decisions of non-conflicted papers before they are released or not.

To complicate the situation further, there are five kinds of roles with distinct privileges: administrator, chair, PC member, external reviewer, and author. External reviewers have some of the privileges of PC members, but what they can see and do is more limited.

These restrictions are implemented through hundreds of PHP conditionals in various parts of the code. HotCRP uses two distinct mechanisms to implement the security policy:

1. HotCRP modifies queries based on the conference settings and the role of the user making the request. For example, a script that allows users to search

through submitted papers adds different joins and selection predicates to the query depending on whether the user is a PC member, an external reviewer, or an author.

2. Some parts of the security policy involve filtering in the application. That is, the application reads sensitive data from the database, but does not show that data to the user. For example, figure 7-4 shows part of the canViewReview function, which determines whether a user can see a given review. There are over two dozen functions like this, and it appears that the HotCRP author has made an effort to centralize security decisions by placing these functions in a single policy module.

Despite the attempts to centralize security decisions in HotCRP, checks are still required in many places; section 7.2.4 describes some information leaks in the system that resulted from missing checks. Essentially, these bugs resulted from HotCRP's failure to provide *complete mediation* [127]. The IFDB model can help, since it ensures that leaks cannot occur except via explicit declassification. Thus, an omission might result in incorrect behavior, but it should not result in a security bug.

A separate concern is that the security policy itself is inherently complicated and therefore might be wrong. It is not clear how helpful information flow control could be at untangling the policy expressed in the canViewReview function in figure 7-4, for instance. While canViewReview could leak information by returning true in a case where it should return false, an IFDB authority closure could just as easily leak information by declassifying inappropriately. Section 7.3.2 argues that DIFC provides a better and perhaps less error-prone to reason about security policies. However, DIFC does not eliminate inherent complexity in the policies.

Figure 7-5 describes a small part of the HotCRP schema, which stores information about users, papers, and reviews. Other tables contain information such as review ratings, preferences, and conference settings. The system makes extensive use of indexes that enforce uniqueness constraints, but like CarTel, HotCRP does not use foreign key constraints.

7.2.2 Security Requirements

As section 7.2.1 shows, the security policy in HotCRP is complicated. In the prototype implementation that uses IFDB, DIFC protects several important types of sensitive information with interesting policies; information flow policies for the remaining

```
function canViewReview($prow, $rrow, &$whyNot = null, $ignoreForceShow = false) {
    global $Conf;
    // fetch paper
    if (!($prow = $this->_fetchPaperRow($prow, $whyNot)))
        return false;
    // policy
    $forceShow = isset($_REQUEST["forceShow"]) && $_REQUEST["forceShow"]
        && !$ignoreForceShow;
    $rrowSubmitted = (!$rrow || $rrow->reviewSubmitted > 0);
    $pc_seeallrev = ($this->isPC ? $Conf->setting("pc_seeallrev") : 0);
    if ($this->privChair && $forceShow)
        return true;
    if (($prow->timeSubmitted > 0
         || defval($prow, "myReviewType") > 0)
        && (($prow->conflictType >= CONFLICT_AUTHOR
             && $Conf->timeAuthorViewReviews($this->reviewsOutstanding
                 && $this->isReviewer)
             && $rrowSubmitted)
            || ($this->privChair && $prow->conflictType == 0)
            || ($this->isPC
                && $prow->conflictType == 0 && $rrowSubmitted
                && $pc_seeallrev > 0
                && ($pc_seeallrev != 3 || !defval($prow, "myReviewType")))
            || (defval($prow, "myReviewType") > 0
                && $prow->conflictType == 0 && $rrowSubmitted
                && (defval($prow, "myReviewSubmitted") > 0
                    || defval($prow, "myReviewNeedsSubmit", 1) == 0)
                && ($this->isPC || $Conf->settings["extrev_view"] >= 1))
            || ($rrow && $rrow->paperId == $prow->paperId
                && $this->ownReview($rrow))
            || ($rrow && isset($prow->myReviewId)
                && $prow->myReviewId == $rrow->reviewId)))
        return true;
    32 lines elided…
    return false;
}
```

Figure 7-4: The canViewReview function implements the confidentiality policy for paper reviews in HotCRP.

Table	Contents
ContactInfo	usernames, contact information, and passwords
ContactAddress	users' street addresses
Paper	paper names, titles, authors, and acceptance decisions
PaperReview	paper reviews

Figure 7-5: The schema in the base version of HotCRP contains twenty-seven tables. The focus here is on five kinds of sensitive information, which is stored in four tables: user contact information (ContactInfo and ContactAddress), user passwords (ContactInfo), papers (Paper), paper acceptance decisions (Paper), and paper reviews (PaperReview).

kinds of sensitive information have not yet been developed. To further simplify the policy, the DIFC prototype does not support all of the configuration options. For instance, in the original version of HotCRP, the conference chair can specify the circumstances under which PC members can view reviews for non-conflicted papers: any time, after the member has completed her own reviews for that paper, or after the member has completed all assigned reviews. The information flow policy assumes that the third option is intended; the other two options add complexity but don't differ in interesting ways.

The goal in converting HotCRP to use IFDB was to protect four types of information: contact information, passwords, paper reviews, and paper decisions. The intended policies for this information are detailed below. To simplify the exposition, the descriptions of some of the policies omit some details that are supported in the implementation. For example, program committee members, the program chair, and the administrator all have slightly different access permissions for other users' contact information, but the differences are not apropos to this discussion.

— *Contact Information.* Users' contact information, which includes names, affiliations, email addresses, and physical addresses, should only be visible to the respective users and to program committee members. Additionally, program committee members' names and affiliations are available to the general public.

— *Passwords.* As in CarTel (section 7.1), user passwords are used only for authentication and never revealed.

— *Paper Reviews.* Program committee members can see reviews for all papers,

except papers for which there is a registered conflict of interest. The PC member must submit all of her assigned reviews before being allowed to see other reviews. After acceptance decisions are made, the program chair can make all reviews, including conflicted ones, available to PC members. At that point, authors can see reviews for their own papers as well.

— *Paper Decisions.* Program committee members can see the acceptance status of all non-conflicting papers. Authors can see the acceptance status of their own papers once decisions are announced.

HotCRP also supports various anonymity settings for papers and paper reviews. The policy described above presumes that the conference is configured such that neither paper submissions nor reviews are blind. Keeping paper and review authors anonymous would require additional work to protect user identities. In particular, it would be necessary to partition some tables to keep private the mapping between papers or reviews and their corresponding authors.

7.2.3 Securing HotCRP With DIFC

Each HotCRP user fills one or more of three roles: author, program committee member, or program chair. In IFDB, these roles are represented as the author, pcmember, and chair principals, respectively. Users act for these principals as appropriate. (Two additional roles, administrator and external reviewer, are ignored for the purposes of this discussion.)

Each HotCRP user, say Cathy, has a tag cathy_contact protecting her contact information, which is stored in the ContactInfo table. All contact tags are members of the all_contacts compound tag. Account creation and session management are handled by closures, in essentially the same way as in CarTel (see section 7.1.3).

To implement the policy that PC members can see user contact information, the pcmember role has authority to declassify for all_contacts. The PCMembers declassifying view implements the policy that the list of program committee members is publicly available. The view distills the names of PC members from ContactInfo and uses authority for all_contacts to declassify:

```
CREATE VIEW PCMembers AS
    SELECT firstname, lastname, affiliation FROM ContactInfo
        WHERE (roles & ROLE_PC) <> 0
    WITH DECLASSIFYING (all_contacts)
```

Table	Secrecy Label	Contents
ContactInfo	{*user*_contact}	usernames and contact information
ContactAddress	{*user*_contact}	users' street addresses
AuthInfo	{sys:auth}	user passwords
PCMembers	{}	PC names and affiliations (view)
Paper	{}	paper names, titles, and authors
PaperReview	{review_*n*}	paper reviews
PaperOutcome	{paper_*p*_outcome}	acceptance decisions
ContactPrincipal	{}	principal IDs for each user
PaperReviewTag	{}	tag for each paper review
PaperOutcomeTag	{}	tag for each paper decision

Figure 7-6: The confidentiality of HotCRP contact information, passwords, paper reviews, and paper decisions is protected with tags. Each paper outcome and each review has its own tag. Paper authors and other sensitive data are not protected with DIFC in the present prototype. Mappings from usernames to principals, review IDs to tags, and papers IDs to tags are also public.

Each review has its own tag, owned by the author of the review. These tags are members of the all_reviews compound tag, which is owned by the chair role. When the chair releases the reviews to authors, she runs a function that delegates each review tag to the appropriate paper authors. Since PC members are allowed to see non-conflicted reviews after they have completed their assigned reviews, they need to be given authority for the pertinent review tags as well. The ReviewDone closure runs with the authority of the chair role every time a review is completed. It checks whether the PC member who wrote the review has completed all of her assigned reviews, and if so, it delegates the tags for all non-conflicting reviews to her.

Each paper decision also has its own tag, which is owned by the chair role. When the chair writes a decision tuple, she also delegates the corresponding tag to all non-conflicted PC members. When the chair makes decisions public, she delegates each paper's decision tag to the paper's authors.

The labeling scheme is summarized in figure 7-6. Tagging paper decisions and user passwords required partitioning some tables. Previously, decisions were stored in the same table as other metadata associated with papers, and passwords were stored along with other user contact information. The decisions were moved to a new

PaperOutcome table, and passwords were moved to the AuthInfo table. Passwords proved to be particularly problematic, as discussed in section 7.3.4. The PaperReview table didn't need to be partitioned, but it would have if one of the security goals were to protect reviewer anonymity with DIFC. Additional tables were introduced to store the mappings from users to principals, reviews to tags, and papers to decision tags, since applications have no way to determine *a priori* which tags they must add to their labels.

In retrospect, handling some parts of the security policy through authority closures and declassifying views instead of through delegating tags would have led to a simpler and more extensible design. For instance, a drawback of the present implementation is that when the conference settings change, many delegations need to be added and revoked. Rather than having a closure delegate all the appropriate review tags to a PC member whenever she completes her last assigned review, PC members could access reviews through a closure that first checks whether the requester has completed her reviews. This proposal is essentially advocating an access control policy, but as section 4.6 discusses, some security policies can be expressed more naturally with access control than with information flow control.

7.2.4 Bugs Prevented

A previously unknown privacy leak in HotCRP was discovered in the process of modifying the system to use DIFC. The bug affected the script contacts.php, which serves two purposes: it shows program committee members the contact information of registered users, and it shows the general public a list of the program committee members and their affiliations. The logic for these two tasks was confused, so anyone could request contact information (including name, email address, and street address) for any registered user with a specially crafted URL. IFDB prevented the bug: when the script read sensitive contact information it was not able to declassify, it was not allowed to produce any output.

Eddie Kohler, the author of HotCRP, provided a list of information leaks in past versions of HotCRP. Some of these bugs involved tables that had not yet been protected with DIFC, but two of them involved paper decisions. To provide further validation of the approach, these two bugs were reintroduced, and IFDB prevented both of them from leaking information.

One bug allowed PC members to see decisions for their own papers prematurely. The bug took advantage of the script that provided the capability to sort papers based

on various attributes. Although PC members could not see the status for conflicted papers, they could sort papers by status, with the accepted papers listed first and the rejected papers last. Thus, they could determine the status of their own papers by checking whether those papers appeared at the top of the list, among the accepted papers, or at the bottom of the list, with the rejected papers.

Eddie Kohler found this to be a subtle bug, but with Query by Label, it is hard to make the mistake that led to the leak. In IFDB, the correct implementation of the sort feature adds to the process secrecy label only the tags that the process is allowed to declassify – that is, the tags for paper decisions the PC member isn't conflicted with. Query by Label then ensures that the conflicted papers decisions aren't visible. (Furthermore, Query by Label protects against the bug regardless of whether the application tried to read the decisions and perform the sort itself, or whether the sorting was done in the DBMS with an `ORDER BY` clause.) An incorrect implementation might add the wrong tags, but then it would lack the authority to declassify, and therefore it would not be able to release anything.

The second bug allowed non-PC-members to see the decisions for their own papers prematurely through the search feature. The search tool allowed program committee members to search through all papers, and it (dubiously) allowed authors to search through papers they had submitted. One available search term was the paper decision. The ability to search based on paper decision should have been restricted to the program committee, but it was not. In the IFDB version of HotCRP, non-PC-members have no authority for any of the tags protecting the acceptance decisions until the chair makes results public. Therefore, they are unable to see the information, despite bugs in the search code.

7.3 Discussion

This section identifies the lessons learned in the process of converting CarTel and HotCRP to use IFDB and PHP-IF. The Query by Label approach was effective at preventing information leaks and reducing the amount of trusted code in the applications. Furthermore, it was easy to identify and implement appropriate labeling schemes to protect sensitive information. Section 7.3.4 identifies areas where future extensions to the model might make it even easier to secure programs like CarTel and HotCRP with DIFC.

7.3.1 Reducing the Trusted Base

The original implementations of CarTel and HotCRP, like most database-backed applications today, placed full trust in the application. By enforcing the security policy in the platform, IFDB reduced the amount of code in CarTel and HotCRP that must be trusted to protect sensitive information. Additionally, specifying the appropriate policy was easier and less error-prone with DIFC than it would have been with access control. For example, there was no need to anticipate the fact that Alice's drives might flow from the Drives table to the LastDrive table via a trigger; in the absence of any declassification, anything derived from Alice's drives will be protected by the alice_drives tag. Moreover, an access control policy alone couldn't prevent bugs in procedures like load_drives from leaking Alice's current location. In an access control system, the PHP code would have to be rewritten as a stored procedure or sandboxed somehow.

Of course, even with DIFC, the security policy must be correctly specified. Programmers express information flow policies programmatically in IFDB, in terms of use of authority. To ensure that sensitive information is protected, trusted code must perform three tasks faithfully: it must label information, it must authenticate users properly to ensure that code runs with the right authority, and it must delegate and use authority judiciously. Abstractions in IFDB helped to simplify these tasks for CarTel and HotCRP and reduce the chance of policy errors as follows:

— *Authentication.* Applications authenticate users in an application-specific manner. IFDB does not provide any authentication primitives, but it does provide a fail-safe default. Scripts that do not authenticate through an application-provided authority closure are executed with no authority. This design choice nullified the authentication bugs in CarTel.

— *Labeling Inputs.* Applications must correctly label sensitive information when it enters the system. However, coming up with appropriate labels in CarTel and HotCRP was easy compared to the task of understanding all the ways in which that data is used. Furthermore, label constraints and triggers can serve as additional checks that data are labeled correctly. For example, in HotCRP, IFDB can enforce the requirement that all reviews must be labeled with subtags of the all_reviews compound tag.

— *Use of Authority.* The fact that Query by Label requires label changes to be

explicit prevents applications from violating the information flow policy inadvertently. For instance, the contact list bug in HotCRP allowed a malicious user to coerce the application into issuing queries for any user's contact information, and Query by Label prevented this attack. Additionally, PHP-IF authority closures, stored authority closures, and declassifying views helped limit the amount of code that must run with authority.

7.3.2 Reasoning about Data Security

DIFC provides an effective way to reason about security concerns for the information stored in the database. Using DIFC required identifying categories of sensitive data, such as drives, locations, and contact information, and determining where those tags should appear in the database. For each query, it was necessary to consider the label the results ought to have, which is helpful in understanding what kinds of sensitive data the application is processing. Furthermore, the fact that IFDB makes declassification explicit drew attention to parts of the program that could potentially create vulnerabilities.

These observations are subjective and based on the author's experiences working with CarTel and HotCRP. Future work could provide stronger empirical evidence of the claim that reasoning about security policies through information flows is less error-prone than other methods. Section 7.2.1 points out that policies in HotCRP are particularly complicated, but only a fraction of the full policy was reimplemented in terms of DIFC.

A related question is whether the code and information flow policies would be done differently for an application designed to use IFDB from the beginning. There is evidence that some parts of the application would likely be different. For instance, HotCRP was designed to always fetch a user's password whenever it loads that user's contact information. That design choice is inconvenient in IFDB, since a process that has read a password would be contaminated by the password label. Nevertheless, the author's goal was to base the information flow policies for CarTel and HotCRP on first principles to the extent that it was practical, so the policies would be the same as in a clean-slate design. The essential underlying principle is that a computation should not read more sensitive information than it needs to do its job, and the Query by Label model with a well-designed authority structure helps enforce this. The CarTel and HotCRP implementations for IFDB generally adhere to this principle, just like they would if they were implemented from scratch.

7.3.3 Schema Decomposition

A few tables in CarTel and HotCRP had to be vertically partitioned, since they contained different kinds of sensitive data. For instance, passwords in both applications were stored alongside other user data. However, most data were already stored in separate tables according to their security concerns, which validates the assertion articulated in section 4.2 that tuple-granularity labeling is a good choice. In the cases where partitioning was needed, views consisting of outer joins (see section 4.1.2) were a useful way to minimize the number of changes to the application.

Outer joins proved to be particularly useful in HotCRP, which frequently did many-way joins that included sensitive tuples, then decided later what the user ought to be able to see. Since many scripts were (dubiously) reading sensitive data that they would never use, the dilemma was how to ensure that PHP processes did not become too contaminated to release anything, ideally without making extensive changes to the application. It would have been inconvenient to have one kind of query for the program chair, another for PC members, a third for authors, and a fourth for external reviewers. Instead, these scripts used outer joins in their queries, so IFDB simply produced null values in place of the fields that were more sensitive than the process label.

7.3.4 Model Extensions

The IFDB DIFC model is straightforward and powerful; most of the security requirements for CarTel and HotCRP were easy to express. However, some parts of CarTel and HotCRP proved to be harder to secure than others, and extensions to the model could simplify some of those tasks. In particular, the experiences with these applications motivate three kinds of extensions:

Boxes and Session State

Passwords in HotCRP proved to be problematic because HotCRP was designed to access all user information, even passwords, via Contact objects in PHP. When a Contact object is initialized, it loads all of the user's information, including the user's password, from the database. Since PHP-IF does information flow tracking at process granularity, it was therefore impossible to construct a Contact object without contaminating the secrecy label of the entire process with the password tag. To solve the problem, a closure loads the password and places it in an Authenticator object

before declassifying. The Authenticator stores the password, but provides no way to access it, except via a checkPassword method that can only be invoked once per PHP-IF instance.

The Aeolus model, which IFDB is based on as described in chapter 3, includes an additional abstract data type called a *box*. A box encapsulates sensitive data; applications can load and store boxes without becoming contaminated, provided that they do not "open" the box to access the data. Therefore, boxes solve the password problem in HotCRP more directly. Additionally, many web applications (not including CarTel or HotCRP) store sensitive data in the web server's session state cache. Placing session state objects in appropriately labeled boxes would help prevent unsafe information flows via session state.

Dynamic Tag Groups

Applications use IFDB's compound tags (section 3.1) to group related tags and refer to them as a unit. Compound tags are important for efficiency. For instance, the CarTel module that produces traffic reports from recent drives would simply add the all_drives compound tag to its secrecy label, rather than adding thousands of individuals' drive tags. Compound tags are static; when a tag is created, the creator must specify one or more compound tags that the tag belongs to. However, experiences with HotCRP have shown that they are static groupings aren't adequate for some policies.

HotCRP allows program committee members to see various information about papers, such as reviews of those papers, provided that the PC member does not have a conflict with the paper. Static compound tags cannot capture notions such as, "all paper tags except the ones for papers Bob is conflicted with." Thus, some processes, such as the ones that run searches over papers, require large labels. Furthermore, these processes must run additional queries to fetch the right set of tags to use.

Dynamic tag groups could make such computations more convenient and efficient by representing the set of tags for Bob's non-conflicted papers directly. Each PC member's set of non-conflicted papers might be large, and there will be one such set for each PC member. Therefore, it is important that dynamic tag groups can be computed on-the-fly. For example, dynamic tag groups might be defined as the result of a query that produces a set of tags. However, dynamic tag groups introduce additional information flow issues: what happens when a tag group changes while a process has that tag group in its label? Nevertheless, it seems important to be able to

express these kinds of groupings, so working out the appropriate semantics would be an interesting direction for future work.

Policy Extensions

Most of the unsafe data processing tasks in CarTel and HotCRP were encapsulated by authority closures or declassifying views. These abstractions provide extensibility: a sufficiently authoritative closure or view can implement any required security policy. However, relying too much on application code to enforce security negates the benefit of using DIFC. Mistakes in these views and closures could lead to security compromises, so more direct support for common policies could improve security further.

Both CarTel and HotCRP require *temporal* security policies, and direct support for these kinds of policies is a promising direction. CarTel distinguishes past drives from current car locations, and HotCRP paper reviews and decisions become available after certain events have occurred. Presently these policies are implemented with closures, but some of the policies are complicated. For instance, the closures that produce Alice's drives from her location data might inadvertently reveal her current location if they declassify data for a drive that is still in progress. In fact, the present implementation is vulnerable to this problem; explicit time limits aren't included in the design. There has been some theoretical work on how to express temporal security policies directly in an access control system [7, 8], but finding a practical solution for DIFC is an open problem.

Chapter 8 Implementation

An IFDB deployment consists of the IFDB database system along with one or more application environments. The application environments and IFDB work together to track information flows and control what information can be released from the system. Additionally, both application code and PL/pgSQL stored procedures can use the Aeolus API (chapter 3) to manipulate the process label and delegate and revoke authority, as shown in figure 2-2 on page 26.

This chapter begins by describing the prototype implementations of the DBMS and the authority state (sections 8.1 and 8.2), then the interface between the database and application environments (section 8.3), and finally the application environments themselves (section 8.4). Section 8.5 covers some of the covert channels that may arise in the implementation, and how they can be mitigated. Finally, section 8.6 identifies ways to reduce the trusted computing base to protect against security bugs in the implementation.

8.1 The Database Implementation

The IFDB implementation is based on PostgreSQL 8.4.10, and relative to PostgreSQL, it includes about 6,300 new or modified lines of C code and 250 lines of PL/pgSQL stored procedures. Tuple secrecy and integrity labels are stored along with each tuple in new, immutable system columns called _label and _ilabel, respectively. The database system was extended in a number of ways to support Query by Label, stored

procedures, declassifying views, new semantics for constraints and transactions, and the Aeolus API.

8.1.1 Query by Label

Chapter 4 introduces two basic rules for queries and updates: rule 4.1 (page 45) states that database queries operate on a subset of the database consisting only of tuples whose labels are no more restrictive than the labels of the process, while rule 4.2 (page 46) requires that all tuples written by a process have the label pair of that process. The IFDB prototype enforces these rules at the point where tuples are read from or written to tables. This approach avoids extensive modifications to the modules responsible for planning and executing queries, which are among the most intricate parts of PostgreSQL. Furthermore, bugs in those modules (for instance, a mistake in the query optimizer) are unlikely to cause information leaks, because the information flow rules are enforced at a lower level.

A tuple is invisible to a process unless its label pair is no more restrictive than the label pair of the process. When IFDB reads a tuple from a table, it compares the tuple labels to the process labels, and pretends that the tuple is not there if the process should not see it. For updates and deletes, new checks were added to throw an exception if a process attempts to write a tuple with a less restrictive label. (If the tuple label pair is more restrictive, then the process will not see the tuple when it determines which tuples to write anyway.) All tuples a process writes are assigned the label of that process.

Adapting the MVCC Infrastructure

Much of the infrastructure required to limit tuple visibility is already present in PostgreSQL as part of the support for multiversion concurrency control (MVCC) [122]. The fact that this infrastructure already exists greatly reduced the effort required to implement Query by Label, so this section briefly reviews the implementation of MVCC in PostgreSQL and explains how it helped simplify the IFDB prototype.

Multiversion concurrency control isolates transactions by assigning a snapshot to each transaction when it starts, and ensuring that each transaction sees only versions of tuples that are present in that snapshot. In particular, queries in a transaction will not see a tuple that was deleted before the transaction's snapshot or inserted after the transaction's snapshot. An update is treated as a delete of the old tuple followed

by an insert of a new tuple. Deleted tuples are retained as long as they are visible in the snapshot of any active transaction, but after that they are garbage-collected by a periodic *vacuum* operation.

The label checks for reads simply extend the existing tuple visibility checks that were originally intended to be used for MVCC. Therefore, if the MVCC implementation is correct, it is highly likely that IFDB properly hides tuples that are too contaminated to appear in query results. The autovacuum daemon, which runs periodic vacuum operations to remove dead tuples, is exempt from the label checks; it sees all tuples.

Indexes and Labels

When PostgreSQL executes a query, it can access tuples in a table in two ways: table scans and index scans. Table scans read each tuple in the table sequentially. First the DBMS applies the visibility checks described in the preceding paragraphs, and then it processes any other selection predicates, including explicit label predicates (section 4.3.3). Index scans use an index structure, such as a B-tree, to identify tuples of possible interest. Then those tuples are read from the table, and the same visibility checks and any remaining predicates are applied.

The IFDB prototype does not provide the necessary index methods to perform the subset comparisons required for Query by Label. Instead, IFDB always checks tuple labels when the tuples are read from the table. The lack of such indexes could introduce performance problems (as well as covert channels) if many tuples match the query but don't have the right labels. However, the author's experiences indicate that applications are typically not designed in such a way that this is a problem. Instead, Query by Label is used as a safety net to protect against privacy and integrity bugs. For instance, a typical query in a medical information system might ask for records about patient Alice, and these queries should use indexes that include Alice's name or patient ID. After the relevant tuples are looked up via the index, IFDB will apply the Query by Label rules to ensure that tuples are only returned to the process if they have appropriate labels. Performance would degrade if instead the application set its secrecy label to {alice_medical} and requested *all* patient records, expecting the database to limit the results to only tuples with secrecy labels {alice_medical} or {}. However, this seems like a strange way to design an application.

For applications that need explicit label predicates (section 4.3.3) to be fast, IFDB does allow labels to be indexed in one important special case. Specifically, index

methods that enable *exact* label comparisons are supported. For the more general subset comparisons required by the Query by Label model, index support could be added in a future release if a need is demonstrated. Labels are sets, typically with a small number of elements (rarely more than two), so an inverted index would be an appropriate data structure. Other data structures for set-valued indexes, such as RD-trees [66], have been studied; however, they are best suited for larger sets where the domain of the elements is small. In IFDB, in contrast, the sets are small, while the number of distinct elements (tags) is very large. Additionally, RD-trees do not support efficient subset queries.

8.1.2 Stored Authority Closures

There are two types of stored procedures in IFDB: ordinary stored procedures run with the caller's authority, while stored authority closures run with bound-in authority. Stored authority closure invocation is different from ordinary stored procedure invocation, as described in section 4.4; when an authority closure returns, the process labels are merged with the labels the process had at the start of the closure call.[1]

The implementation of authority closures includes syntactic extensions to specify the principal bound to a closure when it is created, as well as changes to the PostgreSQL Function Manager to handle the principal and label changes that occur on each closure invocation and return. When a stored authority closure is called, IFDB saves the current label and principal of the process and executes the body of the closure. When the function returns, either normally or via an exception, IFDB restores the previous principal and merges the current and saved labels in the manner described in section 4.4.

8.1.3 Declassifying Views

Declassifying views are implemented through stored authority closures, while endorsing views are not yet implemented. A declassifying view is simply a query that calls a closure to produce the contents of the view. The closure may add secrecy tags to the process label, read the appropriate base tables, and then declassify. While

1. PostgreSQL supports an analogue to authority closures for access control, namely, SECURITY DEFINER stored procedures. However, these procedures cannot execute with less authority than the principal that created them, and they do not merge labels on return.

simple, a significant disadvantage of this approach is that the PostgreSQL query optimizer isn't able to optimize across procedure call boundaries. This means, for instance, that a query for Alice's records in the ContactBilling view (figure 4-3(c) on page 57) would require a table scan, even if the underlying Contact table has an index on the name field.

A future implementation might include direct support for the WITH RELABEL syntax described in section 4.5. The semantics of relabeling views are designed to make such an implementation practical. Recall that relabeling views transform the *process* label; given the label used to query the view, the relabeling function bound to the view produces a new label that can be used to query the underlying tables and produce the contents of the view. This means that queries over these views can be evaluated using the existing Query by Label support, just with a different label. An alternative semantics would be to transform the labels on the *tuples* in the base tables to produce new labels for the corresponding tuples in the view. This approach isn't practical to implement; the DBMS would have to read all potentially matching base tuples and potentially pass them through several layers of view definitions before deciding which ones are visible to the process. In particular, it would have to read tuples whose secrecy labels have tags that are neither in the process label nor within the authority of the view to declassify. Code embedded in the view definitions might leak these tuples, and the fact that the DBMS allowed the tuples to be read might create concurrency conflicts that lead to covert channels.

A more sophisticated implementation of declassifying views that supports the WITH RELABEL syntax directly would need to take care that the query optimizer does not introduce information leaks for certain queries. Normally the optimizer has free license to reorder selections and joins across the view definition. For example, consider the following query over the PCMembers view defined in section 7.2.3:

```
SELECT * FROM PCMembers WHERE firstname = 'Alice';
```

Recall that PCMembers is a declassifying view that selects and declassifies all the program committee members in the ContactInfo table. If the ContactInfo table has an index on the firstname field, the optimizer might produce a query plan that first looks up the users named Alice in the ContactInfo table, then filters out the ones who are not program committee members. In this case, the optimization is harmless, but now consider the following query:

```
SELECT * FROM PCMembers WHERE attack(address) = 0;
```

The attack procedure in this query might similarly be evaluated before the non-PC-members are filtered out, and thus it gets to read sensitive data (users' addresses) that the process should not be able to see. By taking actions such as aborting based on what it read, it could leak information. This is a problem for view-based access control as well, and there are various ways to address it. PostgreSQL 9.1 adds a security_barrier option for views that prevents the planner from pushing the evaluation of functions into a view unless the functions are side-effect free. The present IFDB implementation of declassifying views is not vulnerable because, as discussed, it uses a simpler strategy based on stored authority closures.

8.1.4 Constraints

Chapter 5 describes three types of constraints that IFDB handles differently from ordinary relational database systems: uniqueness constraints, foreign key constraints, and constraints that are enforced by triggers. In fact, all constraints in PostgreSQL are enforced by triggers, but the triggers that implement uniqueness and foreign key constraints are C procedures that are built into the DBMS.

User-defined triggers are simply ordinary stored procedures or stored authority closures that happen to be invoked in response to writes; they do not need to be handled as a special case. As explained in section 5.4, however, triggers may be deferred until the end of the transaction, and deferred triggers, both user-defined and intrinsic, require special handling. IFDB runs deferred triggers with the labels the process had at the time of the action that caused the trigger to fire, rather than the process labels at the time of the commit.

To prevent covert channels due to uniqueness constraints, IFDB uses polyinstantiation. Polyinstantiation requires no special support; the triggers that check for uniqueness constraint violations run with no special authority, and they are bound by the same information flow rules as application code. If the trigger does not observe a duplicate tuple given its label but one exists with a higher label, then polyinstantiation occurs. When polyinstantiation arises in a table, supposedly unique index entries in that table may contain multiple entries identical keys. However, PostgreSQL is already prepared to handle that eventuality since it supports multiversion concurrency control, described in section 8.1.1.

Foreign keys require additional extensions to support DECLASSIFYING and ENDORSING clauses for insert and update statements. Recall from section 5.3.1 that in order for a process to add a tuple a that references a tuple b, the process must

explicitly declassify every tag in $a.L_S \ominus b.L_S$ by adding a DECLASSIFYING clause to the statement. When a DECLASSIFYING clause is present in a statement that inserts or updates a, IFDB checks that the process has authority to declassify for each tag in the set of tags D specified in the clause. Subsequently, when the constraint check is carried out, IFDB temporarily changes the process label to $a.L_S \cup D$, and attempts to read b. This ensures that $b.L_S \subseteq a.L_S \cup D$, and therefore $b.L_S \setminus a.L_S \subseteq D$. Additionally, it verifies that $b.L_S \supseteq a.L_S \setminus D$, and thus that $a.L_S \setminus b.L_S \subseteq D$. Since $a.L_S \setminus b.L_S \subseteq D$ and $b.L_S \setminus a.L_S \subseteq D$, it follows that $a.L_S \ominus b.L_S = (a.L_S \setminus b.L_S) \cup (b.L_S \setminus a.L_S) \subseteq D$, which is the property required by the foreign key rule. ENDORSING clauses would use analogous rules for integrity labels, but they are not implemented.

8.1.5 Transactions

IFDB must do additional tracking to handle the rule for commits (rule 6.1 in section 6.1), which requires that the label pair of a process when it commits a transaction must be no more restrictive than the labels of each tuple in the transaction's write set. Rather than track the write set $\mathcal{W}$ explicitly, IFDB maintains the union of the secrecy labels for tuples in $\mathcal{W}$ as each tuple is written. When a commit is requested, IFDB compares the commit label to this union. Similarly, for integrity labels, IFDB tracks the intersection of the integrity labels of the tuples in $\mathcal{W}$.

PostgreSQL supports two forms of partial rollback: save points and subtransactions. When a transaction rolls back to a save point or aborts a subtransaction, some tuples may be removed from the transaction's write set. In this case, the union of the secrecy labels and the intersection of the integrity labels must be reverted as well. Partial rollbacks are safe: when a transaction commits, only the changes that were not rolled back become visible to other transactions, so it is only those changes that are relevant from an information flow perspective.

8.1.6 Information Flow API

In order to allow stored procedures to change the process label and the authority state, IFDB implements the Aeolus API, described in figure 3-1 on pages 32 and 33, as a set of built-in functions. IFDB also supports a few extensions to the Aeolus API. New functions are provided for convenience, such as getSecrecyLabel and getPrincipal, which return the current secrecy label and principal, respectively. Additionally, IFDB

extends the makeSubtag function to take a set of compound tags instead of a single compound tag. This change allows tags to be members of multiple compound tags, should the need arise.

Section 8.1.2 describes the implementation of stored authority closures, which are similar to Aeolus authority closures. Section 8.1.3 covers another important enhancement to the model, declassifying and endorsing views. Both of these features involve syntactic extensions. An ordinary built-in function handles reduced authority calls. Applications must specify the stored procedure to call with reduced authority by giving the name of the procedure as a string, since PostgreSQL does not support higher-order functions.

8.2 The Authority State

The authority state consists of all of the tags and principals in the system, as well as the acts-for links and grants among them. The Aeolus platform uses a distinguished authority server to store the authority state. In contrast, IFDB deployments store the authority state in the same database as application data, which simplifies and speeds up the implementation. If the authority state were stored on a separate server, then creating a principal for a new doctor in a medical clinic and recording contact information for the doctor would require a distributed transaction, for instance.

Acts-for links and grants are stored in system tables pg_actsfor and pg_grants, respectively. The tables can only be written through the functions that implement the Aeolus API. Indexes are used to make lookups efficient. To check whether a principal p has authority for a tag t, IFDB searches the authority graph for a path from t's owner to p. IFDB does not implement the Aeolus system's optimization of memoizing the results of these searches. For the applications studied in chapter 7, paths in the authority graph tend to be short, and the cost of searching in the graph is small compared to the total cost of running a database query. The PHP-IF implementation does include support for memoization, as mentioned in section 8.4.1.

Similarly, compound tags assignments are stored in the pg_compounds system table. An index provides a fast way to look up a list of compound tags that contain a given tag. (Typically tags are members of a single compound tag, if any.) Compound tags require special treatment in two places. First, when a process declassifies a compound tag, IFDB removes any secrecy tags that are subtags of that tag from the process label. Second, the subset comparisons that are needed to implement

Query by Label take compound tags into account. In determining whether a label L_A is a subset of label L_B, IFDB first compares the labels without checking the compound tag membership. Any remaining tags in L_A that were not found in L_B are then checked against L_B's compound tags.

There are two special cases for compound tags where the behavior that IFDB implements differs from what one might expect. One is the case where a process declassifies a tag that is a member of a compound contained in the process label. Since the compound tag is a placeholder for a set of tags, it is tempting to think that the resulting process label should contain all subtags of the compound, except for the one that was declassified. However, that could be a very large set, and IFDB provides no efficient way to represent it. The second corner case is label comparisons where one label L_B contains *all* of the subtags of a compound tag in another label L_A. IFDB does not waste time to detect this unusual situation, so it will never conclude that $L_A \subseteq L_B$. Neither of these cases correspond to situations that seem likely to arise in practice.

The tuples that comprise the authority state are restricted to have empty secrecy labels, so all Aeolus operations that modify the authority state require that the caller have an empty secrecy label. This restriction ensures that processes cannot signal information through modifications to the authority state. In IFDB, even the parts of the platform that do authority checks are constrained by the information flow rules, so it would be possible to support secret acts-for links and grants. However, authority state modifications are rare; they are typically performed when a new user is added to the system, for instance. Thus, it isn't clear that it is worth complicating the semantics to support non-empty secrecy labels on the authority state.

IFDB caches information from the pg_actsfor, pg_grants, and pg_compounds tables in hash tables to avoid having to go through the normal query processing code to do authority lookups. It uses the same caching mechanism that PostgreSQL uses to cache important, frequently accessed information such as the database schema. Whenever part of the authority state is updated, IFDB invalidates the corresponding mappings in the cache.

8.3 The Database Interface

When an administrator sets up an IFDB deployment, she configures the database to allow only trusted DIFC runtime environments to connect. These environments

run application processes and track information flows in those applications. They coordinate changes in the label and authority of the running process with IFDB. This section describes how the application environments and the database system interact with one another.

8.3.1 The Frontend/Backend Protocol

The DBMS and the application environment both keep local copies of the current label and principal of each application process. Changes in the label and principal are communicated by means of the low-level protocol that PostgreSQL uses to send queries from the client (the *frontend*) and retrieve result sets from the DBMS (the *backend*). IFDB adds a new IFCUpdate protocol message that is sent from the client environment to the DBMS whenever a label change occurs (for instance, when the application adds a secrecy tag or declassifies), or when the running principal changes, for instance due to an authority closure call or reduced authority call. IFCUpdate messages are also sent from IFDB to the client when principal or label changes occur due to the activity of stored procedures.

Principal and label changes are sent lazily, along with the next command or result. Thus, multiple changes can be coalesced into a single message, and IFCUpdate messages are always piggybacked on other messages. To ensure that it is impossible for the DBMS and the client to change the process label without being aware of each other's changes, all of the protocol messages IFDB uses are synchronous, and IFDB assumes that client applications are single-threaded.

8.3.2 The Client Library

An extended version of PostgreSQL's libpq client library provides a simple C interface for trusted DIFC runtime environments, hiding the messy details of the protocol. Applications call the library to get and set the process labels and current principal. The library stores the labels and principal locally and sends any updates to the database along with the next command.

The libpq interface is intended to be called only by parts of the DIFC implementation, so it does not perform any permission checks. Instead, those checks are the responsibility of the caller. This design makes it possible for DIFC runtime environments to implement new information-flow abstractions.

8.4 Clients

The PHP-IF and Python-IF application platforms are PHP and Python implementations of the Aeolus model, which was originally developed for C# and Java [17, 18]. Additionally, they interact with IFDB to implement Query by Label. Since the emphasis of this dissertation is on the DBMS, the client implementations are not as robust as Aeolus or IFDB; the clients are missing some features and useful optimizations.

Three other clients were also adapted to work with IFDB. The psql tool provides a command-line interface to the database, mainly for administrative purposes. It was extended to allow the administrator to set the label and principal of the psql process to support debugging and administrative operations. The pg_dump and pg_restore tools, which make it possible to back up the database, were also enhanced to include tuple labels in database backups.

8.4.1 The PHP-IF and Python-IF Implementations

The libpq library handles the complexity of the database protocol and label management, so both PHP-IF and Python-IF contain only about 150 new lines of C code to handle the database interaction. The remaining changes to support IFC are implemented at a higher level, in PHP and Python. Each respective implementation is about 1100 lines of code. Most of the new code supports PHP and Python versions of the Aeolus API, including support for authority closures. Calls that change the authority state are simply translated into SQL statements that call the corresponding IFDB built-in procedures. Procedures that change the process label or principal do any necessary authority checks, then invoke lower-level libpq routines to make the necessary changes.

PHP-IF and Python-IF also interpose on output, so programs that are too contaminated can't release information. In PHP-IF, this is accomplished by using PHP's built-in output buffering support. Output from the application that is intended for the user is stored in an internal buffer. The PHP-IF runtime registers a callback that is invoked whenever the buffer is flushed (which happens automatically when the application terminates). The callback checks whether the process is allowed to communicate given its current label; if not, the user receives an error indicating that the process is too contaminated. The Python implementation achieves the same end result by replacing the sys.stdout stream with a file object that performs

the appropriate checks. The prototypes do not yet interpose on other library calls that support I/O, such as the APIs for sending email and accessing the file system. However, this limitation can be addressed in a straightforward way. The necessary file-system semantics have been explored in Aeolus [18], and RESIN [150] considers information flows via email.

The PHP-IF implementation includes a shared-memory cache that stores memoized results of past authority and compound tag lookups. Additionally, when an application changes the authority state, for instance by creating a new subtag or delegation, the mapping is opportunistically stored in the cache. The cache is shared among all PHP-IF instances on any given web server. It is important because most processes perform at least one operation that requires authority, such as a declassification. Caching drastically reduces latency by avoiding extra round-trips to the database for these common operations. Unlike Aeolus, the PHP-IF implementation does not have a cache coherence protocol to maintain consistency across multiple web servers. Since authority state updates are relatively rare, a straightforward invalidation-based protocol like the one used in Aeolus would be a practical solution.

Since much of the functionality provided by PHP-IF and Python-IF is implemented in PHP and Python, it is possible that a malicious programmer could subvert the protection mechanisms. For instance, both PHP and Python make it possible (though awkward) to access instance variables of an object that are not intended to be public. Hence, the present implementations of these clients can only be considered secure against buggy code, not actively malicious code.

8.4.2 Extensions to the Aeolus Model

Typical PHP scripts are executed sequentially, and they intersperse program output, including embedded HTML fragments, with code execution. PHP-IF implements two extensions to the Aeolus model to account for this model of web programming. The extensions help simplify the development of applications in PHP-IF.

Automatic Output Declassification

The Aeolus model requires processes to have empty secrecy labels in order to communicate with the outside world, and therefore processes that have read secret information must explicitly declassify all the tags in their respective secrecy labels.

However, this requirement is too cumbersome for PHP programs, where output and computation are intermixed. In fact, it would be dangerous for processes to declassify to produce output in the middle of a computation, because a process might inadvertently write the uncontaminated data back to the database.

The first extension, automatic output declassification, addresses the problem by allowing a PHP process to register a set of tags that will be effectively declassified when the process produces output for the user. In other words, the process secrecy label does not need to be empty to produce output; it just needs to be a subset of the registered tags. Tags are registered by calling autoDeclassify, which requires that the caller have authority for those tags. The capability provided by automatic output declassification could have been achieved without the new primitive by having applications create authority closures that declassify and produce output. However, providing the feature as a primitive allows applications to continue to use the same output routines as they would if they were ordinary PHP programs. Other information flow systems provide similar functionality, for instance, send labels in Asbestos [85] and gate labels with the $\star$ privilege in HiStar [152].

The switchTo Function

The second extension is the switchTo primitive, which applications use to implement authentication. In Aeolus, authority changes are done via reduced authority calls or closures. One could imagine writing an application-specific authentication closure that runs with authority for all users; if a user authenticates successfully, it performs a reduced authority call to execute the rest of the script with the authority of the user's principal. That strategy won't work for many PHP scripts without substantial changes, though, since PHP scripts are executed sequentially, with much of the computation typically happening outside the context of any function call. The switchTo function supports the PHP programming model by allowing authority closures to grant authority to their callers. A closure that acts for principal p can call switchTo(p) to set the top-level principal – that is, the principal that the process has when it is not running in a closure or reduced authority call – to p.

8.5 Covert Channels

This section discusses potential covert channels that can arise in the implementation. Similar to prior DIFC work, IFDB does not close all possible channels, but this

section does explain the issues involved and identify ways of preventing or mitigating the channels.

8.5.1 Timing Channels

Timing channels are inevitable in any system that permits different trust domains to dynamically share resources, and the IFDB implementation is vulnerable to timing attacks. For example, a contaminated process could try to communicate whether Alice has HIV by running expensive queries that consume large amounts of resources if she does, and running no additional queries if she does not. A collaborator with an empty secrecy label could then notice the amount of time it takes his own queries to complete, and thus determine with some certainty whether Alice has HIV.

Various techniques have been developed to mitigate the impact of timing channels [6, 78, 97], for instance, by quantizing response times. IFDB can incorporate these defenses, but the present prototype does not do so. Instead, this dissertation takes the position that addressing large-scale leaks and buggy (but well-intentioned) code is of primary importance, and it leaves questions about the practicality of timing channel mitigation in databases for future work.

The Query by Label implementation in IFDB gives rise to a channel that might be mitigated in more specialized ways, as opposed to the general techniques studied in prior work. Suppose a process with an empty label attempts to read an HIVPatients tuple for Alice (given the HIVPatients table in figure 5-1 on page 64). Since the process has an empty secrecy label, it will see no results regardless. However, if the tuple does not exist, then the DBMS only needs to read the primary key index, whereas if it does exist, the DBMS must also read the tuple from the table to check its label. Therefore, the query will take more time if the tuple for Alice exists. A good mitigation technique is to store labels in the index, so the number of I/O operations is the same either way.

8.5.2 Allocation Channels

If principals and tags were allocated in a predictable sequence, that sequence might reveal unintended information, such as the order in which papers were submitted in HotCRP. IFDB allocates principal and tag IDs from the output of a cryptographic pseudorandom number generator to avoid this problem.

A more subtle channel involves tuple allocation. PostgreSQL allocates storage

for tuples from per-relation heap files, so the relative order of tuples within a relation can be influenced, to a degree, by the sequence of modifications to the relation. This fact might allow an uncontaminated process to deduce the presence of a high-labeled tuple, for instance, a tuple for Alice in the HIVPatients table.

The channel can be avoided by ordering tuples returned to the application by any deterministic function of their values. Of course, if the application already requested an order with an ORDER BY clause in the query, then no additional sorting is required. As with timing channels, these channels appear to require a concerted malicious effort to exploit, so the IFDB prototype does not force query results to be ordered.

8.5.3 Conflict Channels

Section 6.2 describes conflict channels, which convey information due to conflicts among transactions. The treatment in that section considers data conflicts in the abstract, and not a specific concurrency control mechanism such as locking, optimistic concurrency control [87], or multiversion concurrency control [122]. The rule presented there avoids conflict channels and is applicable to any of these techniques; additionally, appendix A describes an alternative strategy that works for snapshot isolation, but not for serializability.

However, both solutions rely on the assumption that the DBMS does not report *false* conflicts among transactions. For example, if the DBMS uses page-level locks and a particular page contains both public and secret tuples, then processes might be able to leak secret information via conflicts for the page lock. The IFDB prototype uses a snapshot isolation [10] approach with fine-grained write locks and no read locks, so it doesn't suffer from false conflicts.

More generally, however, avoiding conflict channels may require restricting the set of techniques available to the DBMS implementor. For example, a common optimization for lock-based protocols is *lock escalation*, where many fine-grained locks are exchanged for a smaller number of coarser-grained locks. However, lock escalation is mainly used for transactions that read or write many tuples, and it's unclear whether it is even a realistic goal to conceal the activities of large transactions from each other while ensuring serializability. A more easily attainable goal is to prevent conflict channels for online transaction processing (OLTP) workloads, where transactions are short-lived. Both of the applications studied in chapter 7 fall into this category.

8.6 Reducing the Trusted Base

The DBMS itself is a large and complicated piece of software, so there is a potential concern that it might have bugs that undermine security. Such vulnerabilities appear to be relatively uncommon in practice. The implementation of IFDB discussed above adds code to the database, and this code must also be part of the trusted base. It is possible to avoid much of this added code, at the cost of some performance degradation, by implementing IFC through query rewriting.

A proof-of-concept rewriter with 1,200 lines of ANTLR grammar and Java code was built to explore the idea. The rewriter handles the essence of Query by Label, but not the full Aeolus API. It modifies table creation commands to add a label field, and to include the label in primary keys to support polyinstantiation. It transforms queries to add the appropriate subset comparisons on the labels. The rewriter is simple and easy to verify, and with this approach, the DBMS need only be trusted insofar as it faithfully executes the subset of SQL:1999 accepted by the rewriter. Section 11.2.3 explains how the proxy approach could be extended to reduce trust in the DBMS even further. Preliminary measurements show that for a benchmark based on the TPC-C specification with 10 warehouses, performance with the proxy is about 14% lower than with PostgreSQL, but future work might decrease the overhead.

Chapter 9 Performance

Chapter 7 showed that IFDB is easy to use and that it improves security of real applications. This chapter completes the evaluation by showing that IFDB performs well. The main result is a benchmark that shows that CarTel performs about as well under PHP-IF and IFDB as it did under PHP and PostgreSQL. Section 9.2 presents and analyzes this result.

Additional benchmarks based on TPC-C provide further insight into how information flow control affects the scalability of the database specifically. The dominant costs are simply those associated with reading and writing tuple labels. Section 9.3 confirms this by exploring the relationship between database throughput and the size of tuple labels.

The application platform must do extra work to track information flows as well. Application performance with DIFC has been widely studied in prior work [18, 85, 86, 125, 150, 152]; this chapter looks at both database and application performance in macrobenchmarks, but it examines the performance of the DBMS more closely.

9.1 Experimental Setup

The benchmarks described in this chapter were run on a database server with four Xeon E7310 CPUs (16 cores), 8 GB of RAM, a RAID controller with a 256 MB battery-backed cache, and three 15,000 RPM SAS drives in a RAID 5 configuration. The server ran Linux kernel version 2.6.38. The database system in each test was either

IFDB or PostgreSQL 8.4.10 (from which IFDB is derived). Greg Smith's pgtune utility, version 0.9.3, adjusted the PostgreSQL settings for an OLTP workload, taking into account the amount of memory available. Two database parameters were tuned manually: checkpoints_segments was set to 64 to prevent the database log from filling up too quickly, and max_connections was set to 1,500 to allow enough clients to connect simultaneously to achieve peak throughput.

Several web servers were connected to the database via a Gigabit Ethernet switch. The web servers were substantially less powerful than the database server: each had a hyper-threaded 3.06 GHz Pentium 4 CPU and 2 GB of RAM. They ran Apache 2.2.15 and either PHP-IF or PHP 5.3.10. The web servers also used APC 3.1.3p1, which caches compiled PHP scripts. This caching is important, since much of the PHP-IF implementation is written in PHP, and recompiling it on every script invocation would incur a large performance hit.

9.2 Macrobenchmarks

A series of benchmarks involving CarTel demonstrate that IFDB has good real-world performance. The first set of benchmarks, covered in section 9.2.1 are read-intensive. They involve the CarTel web portal, which allows users to view location data for their cars and their friends' cars. Section 9.2.2 presents a write-intensive benchmark that measures how fast the database can process sensor readings.

9.2.1 CarTel Web Portal Performance

The benchmark described in this section compares the performance of the new version of the CarTel website, which runs on IFDB and PHP-IF, to the original version, which uses PostgreSQL and PHP. The database was populated with 18 GB of real data, consisting of 177 million location readings collected from 409 users over a 27-month period. The location readings are representative of what might be observed in a bigger system over a shorter time period, but clearly a large-scale deployment would have more users and cars. However, it isn't clear that this difference is relevant to the cost of using IFDB. In both small and large deployments, the DBMS is likely to find user and car data for active users in the buffer cache most of the time, since those data are small and frequently accessed; the vast majority of the disk I/O comes from accessing past location and drive data.

Freq.	Request	Description
0.50	get_cars.php	location updates (AJAX)
0.30	cars.php	show car locations
0.08	drives.php	show drive log
0.08	drives_top.php	common driving patterns
0.03	friends.php	view and set friends
0.01	edit_account.php	edit personal info

Figure 9-1: The CarTel web benchmark uses a distribution of HTTP requests intended to mimic user activity on the real CarTel website. The get_cars.php script is the most common because it is invoked by asynchronous JavaScript (AJAX) on the client to update the map. In addition to the scripts listed above, login.php is invoked once at the start of every user session.

The benchmark uses a methodology based on TPC-W [139] to measure the maximum sustained throughput of the system. Simulated clients each log in as a random user, make a random sequence of HTTP requests, then end their sessions. The "think time," or duration between two HTTP requests from the same client, ranges from 0 to 70 seconds, following a truncated negative exponential distribution. The length of each session also follows a truncated negative exponential distribution, and can be up to about 60 minutes. The vast majority of think times and session durations are closer to the low end of the range. The actual requests, however, are tailored to the CarTel workload, rather than the fictitious online catalog system that TPC-W specifies.

Following the initial login, simulated web clients request pages according to the distribution in figure 9-1. The distribution is intended to mimic a real workload. To obtain more consistent performance, the benchmark did not generate requests for users who had more than five cars. There were four such users, one of which was a cab company. The load generator adjusted the number of clients to achieve peak throughput while keeping the 90th percentile response time under three seconds, which is the criterion used by TPC-W as well.

Figure 9-2 shows the maximum number of web interactions per second (WIPS) the web servers and database could sustain subject to the constraint on response time. With three web servers, performance was limited by the database, which was disk-bound. Five two-hour trials did not demonstrate a statistically significant

	Web Interactions Per Second	
	PostgreSQL + PHP	IFDB + PHP-IF
database-bound	229.3	230.4
web-server-bound	132.0	103.5

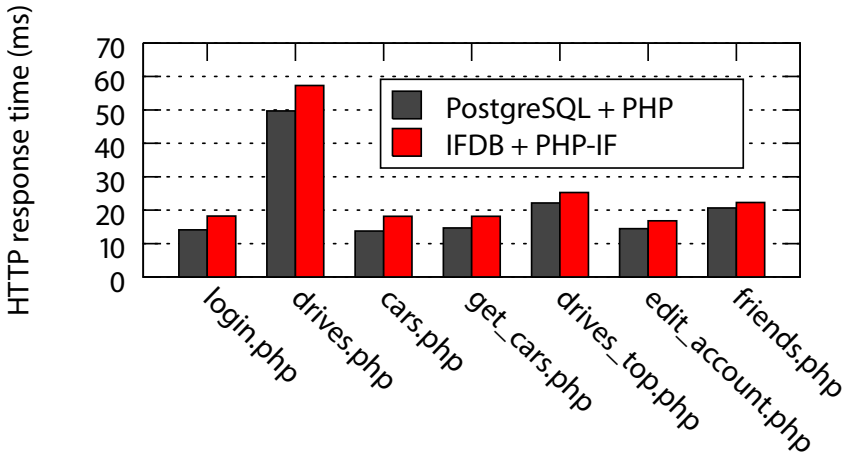
Figure 9-2: The table shows the throughput of the CarTel website, with and without information flow security. The database-bound workload uses three web servers, and the web-server-bound workload uses just one.

difference between IFDB and PostgreSQL in this scenario. With one web server, however, the web server's CPU was the bottleneck and throughput was 22% lower with IFDB and PHP-IF. The results demonstrate that, while information flow control adds some overhead to the web servers, it doesn't perceptibly affect the scalability of the database for a real web application.

Nevertheless, these results raise the question of where the overhead in the web servers comes from. To help answer this question, figure 9-3 reports the HTTP request latency on an idle system, with a single client issuing requests serially. The weighted mean increase in response time with IFDB and PHP-IF was 24%. The highest absolute increase was in drives.php, which had to handle tags for each of the user's friends.

The latency difference for each script mainly reflects the additional time required to look up principal and tag IDs, perform label manipulations to read sensitive data, and check that the process is allowed to release what it read. PHP-IF caches authority information, so these checks generally don't require communication with the database. A separate measurement shows that the additional time required to load the PHP-IF module, which is largely written in PHP and must be initialized at the start of every script, was only 0.2 ms. Results also show that the use of APC (see section 9.1) is important, since APC caches compiled PHP code and therefore avoids the need to recompile the PHP-IF implementation on every script invocation. Without APC, the increase in startup latency was found to be 8.3 ms.

A better implementation of PHP-IF would have lower overhead. The prototype is largely written in PHP itself to save on development time, since the emphasis here is on the performance of the DBMS. Furthermore, the PHP-IF prototype does not implement optimizations, such as a more sophisticated cache, that are present in the Aeolus system it is based on.



Script	Response Time (ms) PostgreSQL + PHP	IFDB + PHP-IF
login.php	14.1	18.3
drives.php	49.7	57.3
cars.php	13.7	18.2
get_cars.php	14.7	18.2
drives_top.php	22.1	25.3
edit_account.php	14.5	16.8
friends.php	20.7	22.3

Figure 9-3: CarTel web request latency on an idle system was substantially higher in the version that used PHP-IF. The increased latency explains the drop in throughput in the web-server-bound experiment in figure 9-2.

9.2.2 Sensor Data Processing Throughput

CarTel also has a *track server* component, introduced in in section 7.1.1, that processes and stores GPS location readings. The web server is not involved in this part of the system. For each reading, a new tuple is inserted into the Locations table, and two triggers fire, which read from the Cars table and update the Drives and LocationsLatest tables. In IFDB, these triggers run as authority closures so that they can read Cars and update Drives and LocationsLatest without contaminating the process performing the insert. Another trigger keeps the LastDrive table updated with information about the most recent drive for each car; since this trigger only reads and writes drive data, it did not require any changes. Figure 7-3 on page 97 illustrates these triggers.

The track server issues 200 inserts to the database per transaction, which improves performance, since each transaction is synchronous and incurs a round-trip message delay. Also, PostgreSQL 8.4.10 lacks support for group commit, so grouping writes manually improves DBMS throughput. In IFDB, each Locations tuple must be labeled with the appropriate user's location tag so that the platform will subsequently protect it from improper release. Each incoming location reading may have a different secrecy label, and it is important for performance that this requirement does not result in 200 separate insert statements. Therefore, labeling is handled by the addlocation stored authority closure shown in figure 7-3. The input to the closure is a sequence of location readings. The closure runs with authority for the all_users principal; for each location reading, it adds the appropriate user location tag to the process secrecy label, inserts the tuple, and uses its authority to declassify the tag.

The benchmark replayed real location readings from the 27-month trace to the database as fast as possible and measured the average throughput. PostgreSQL was able to process 2,479 location readings per second, while IFDB processed 2,439. The 1.6% difference reflects the additional bookkeeping associated with properly labeling the data, as well as the overhead of storing the labels themselves. The following section explores the impact of the latter source of overhead.

9.3 The Cost of Labels

IFDB must store the label pair for every tuple, and compare the labels to the process labels on every read and update. This section explores those costs, independent of the other differences associated with modifying the application to support information flow control.

Intuitively, the cost of labels should be related to the number of tags in the labels, and also the relative increase in the size of each tuple. Labels are typically small: Tuples in CarTel and HotCRP have secrecy labels with 0–2 tags each, and integrity labels were not used. Computations that combine data with many different tags use compound tags (section 3.1), which summarize an arbitrary number of related tags with a single tag.

Nevertheless, to gain a better understanding of the cost of labels, this section evaluates IFDB's performance with tuple secrecy labels ranging from zero tags up to ten tags – more than expected in practice. In each experiment, all of the tuples in the database and all processes had identical labels. Thus, the label comparisons are simple and always succeed; however, the DBMS still incurs some cost to store and manipulate the labels.

The DBT-2 benchmark was used to measure performance. It implements the TPC-C [140] specification, which is a standard way of measuring online transaction processing (OLTP) performance. The TPC-C benchmark simulates an order-processing system for a fictitious wholesale supplier with a number of warehouses. Simulated users order items by placing New-Order transactions, and each user waits a random amount of time, called the *think time*, between requests. Each warehouse serves a fixed number of users, so TPC-C effectively requires systems that support larger transaction rates to use more warehouses, and hence larger databases. A common simplification, however, is to hold the number of warehouses constant and set the think time to zero; the benchmarks presented here do exactly that.

To better capture the distinction between I/O and computational overhead, the benchmark was run on an in-memory database with 10 warehouses and an on-disk database with 150 warehouses. Figure 9-4 reports the results. The transaction rates are scaled so that performance relative to PostgreSQL is directly comparable. Within the range studied, each tag reduces throughput by about 0.6% for the in-memory workload and 1% for the on-disk workload. Since labels with one tag are the most common, 1% appears to be a is a conservative estimate of the overhead of managing labels for real applications in the database.

Much of the overhead comes from I/O and cache pressure. Labels in IFDB increase the size of each tuple by four bytes per tag, with corresponding implications for disk bandwidth and the buffer cache. (The label length is stored in a single byte in the tuple header, which was previously unused for alignment reasons.) For example, Order_Line tuples, which are responsible for the majority of the I/O in TPC-C, are 89 bytes. Each tag adds 4.5% to the space consumed by an Order_Line tuple, and thereby

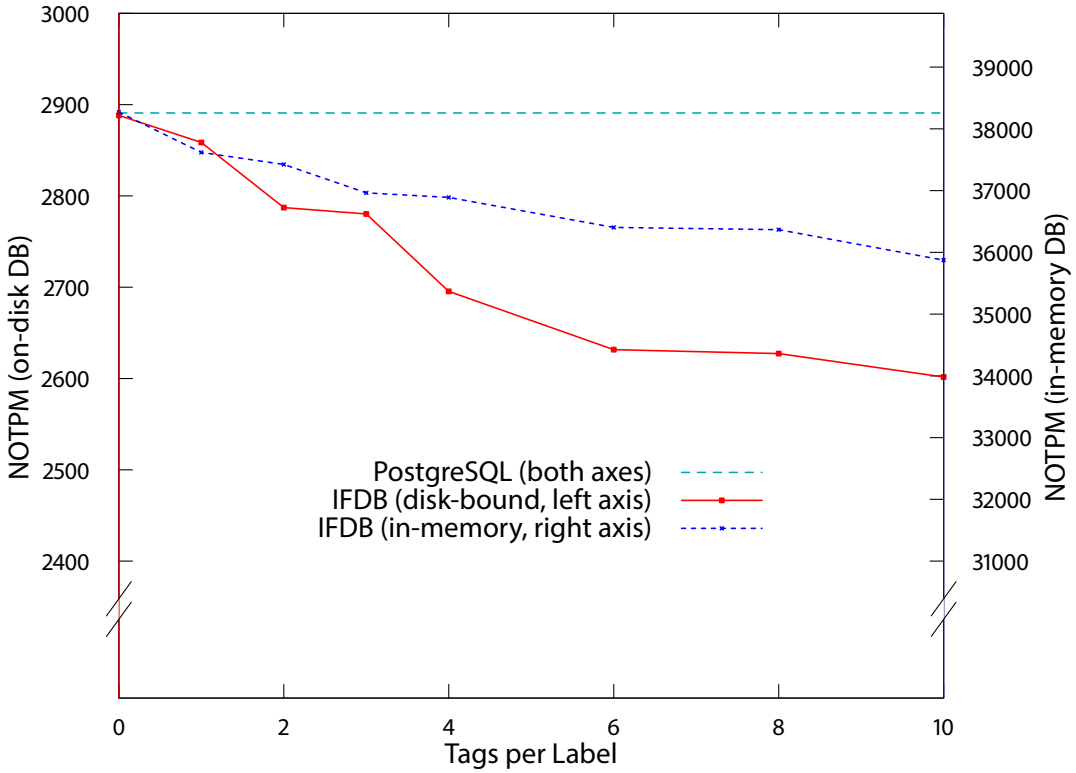


Figure 9-4: The throughput of DBT-2, measured in New-Order transactions per minute, drops slightly for each additional tag that is added to tuple labels. The larger relative drop for the disk-bound database shows that much of the cost is related to reading, writing, and caching the labels. The results are scaled so that the relative differences are directly comparable.

decreases the number of such tuples that can be stored per page. Tuples and their are stored in *heap pages*. Paging statistics indicate that within the range studied, the database read roughly 3.2% more heap pages per transaction for each additional tag in the label. This resulted in a smaller drop in overall performance, however, partly because most OLTP queries involve index reads, and IFDB does not store labels in index pages.

Chapter 10 Related Work

IFDB combines techniques developed in several lines of research. Prior work on information flow control in database systems has looked at a number of problems that affect IFDB, such as how to handle constraints and transaction scheduling without introducing covert channels. However, this work is confined to mandatory security policies that have limited applicability outside of military systems. Separately, abstractions for fine-grained information flow control have been developed for programming languages and operating systems. IFDB builds on these abstractions, and incorporates prior database work where it is applicable.

This chapter reviews the information flow models used by prior database systems and by IFDB, and describes past systems and how they have influenced IFDB's design. Complementary research, such as work on statistical databases, is also covered. This work is important because it provides programmers with the framework they need to define appropriate policies for information release; IFDB provides abstractions to help programmers implement the intended policies securely.

10.1 Information Flow Models

In Bell and LaPadula's seminal work on information flow control [9], data are labeled with classification levels (for instance, *confidential* or *top secret*) and categories (for instance, *nuclear* or *counterintelligence*). In their parlance, IFDB satisfies the simple security property (no read up), strong-$\star$ property (writes only at the current level),

and the weak tranquility principle (level changes are permitted if they are safe, or if they are made with the proper authority). Biba [11] proposes a set of rules for integrity, which are the dual of the Bell-LaPadula confidentiality rules. Denning [32] formalizes the Bell-LaPadula classification levels as a mathematical lattice. IFDB's secrecy/integrity label pairs form a lattice as well, ordered by the less-restrictive-than relation (definition 3.1). The safe information flow rule (rule 3.1) is standard and comes from this work.

Most subsequent work on database security is based on the aforementioned models, and it commonly exhibits two notable characteristics, owing to the models' origins in military systems. First, database systems tend to use *mandatory* security policies, which are systemwide policies defined by an administrator. Second, they are organized around a (typically small) set of levels and categories intended to apply to broad classes of data. Such systems are referred to herein as following the *centralized model*.

In 1997, about two decades after the first models were introduced, Myers and Liskov [113] introduced the Decentralized Label Model (DLM). The DLM permits *discretionary* security policies, wherein individual principals in the system have authority to control the policies for their own data. For example, if Alice and Bob have different doctors, then their respective medical records should have different security policies. The DLM supports this need, whereas systems based on the centralized model do not, due to assumptions about centralized policies and a small number of categories. IFDB is the first database system to incorporate the decentralized model.

10.2 Information Flow Systems

Prior work on mechanisms for information flow control has come from three different camps: the database community, the programming language community, and the operating system community. The approaches in each line of work are often quite different, and IFDB incorporates ideas from all three.

Figure 10-1 compares a representative set of prior information flow systems. There are several important points of comparison:

— The choice of model (centralized or DIFC) is fundamental to the design, as discussed in section 10.1. Like most of the recent work in programming languages and operating systems, and unlike past work in databases, IFDB adopts the DIFC model.

System	Year	Environment	Model	Type	Granularity
ADEPT-50 [31]	1969	OS	centralized	dynamic	processes
Denning [31]	1975	language	centralized	dynamic	objects
MLDS [54]	1986	DBMS	centralized	dynamic	tuples
ASD_Views [55]	1988	DBMS	centralized	dynamic	relations
SeaView [37, 101]	1988	DBMS	centralized	dynamic	fields
LDV [43]	1988	DBMS	centralized	dynamic	fields
Sybase Secure [138]	1990	DBMS	centralized	dynamic	tuples
SINTRA [50, 105]	1991	DBMS	centralized	dynamic	fields
IX [106]	1992	OS	centralized	dynamic	processes
Trusted Oracle [13]	1992	DBMS	centralized	dynamic	tuples
Trusted Rubix [73]	1992	DBMS	centralized	dynamic	tuples
Informix Secure [72]	1992	DBMS	centralized	dynamic	tuples
Jif [112, 113]	1997	language	DIFC	static	objects
Asbestos [85, 141]	2005	OS	DIFC	dynamic	processes
HiStar [152]	2006	OS	DIFC	dynamic	processes
Flume [86]	2007	OS	DIFC	dynamic	processes
DIFCA-J [151]	2007	language	assertions	dynamic	objects
DStar [153]	2008	OS	DIFC	dynamic	processes
Laminar [125]	2009	OS/lang	DIFC	dynamic	objects
RESIN [150]	2009	language	assertions	dynamic	bytes
UrWeb [19]	2010	DBMS/lang	assertions	static	fields
Aeolus [18]	2012	OS/lang	DIFC	dynamic	processes
IFDB	*2012*	*DBMS/lang*	*DIFC*	*dynamic*	*tuples/processes*

Figure 10-1: IFDB's information flow model is based on recent developments in programming languages and operating systems. IFDB also borrows ideas, such as polyinstantiation, from earlier work on trusted databases based on the centralized model for information flow control.

— Information flow properties can be verified statically or dynamically. Static approaches improve robustness by catching errors earlier, but they are harder to program. Static flow analysis is predominantly found in programming languages, but it can also be applied to operating systems (for instance, channel contracts in Singularity [71, 134]) and databases (as in UrWeb [19]). IFDB uses a dynamic approach, but incorporating static analysis techniques is a promising area of future work.

— Dynamically tracking flows at a fine granularity, such as objects or bytes, is expensive. Therefore, many prior systems track flows at the level of processes instead. This dissertation advocates fine-grained, tuple-level tracking in the DBMS and process-level tracking in the clients (PHP-IF or Python-IF).

Each line of work – programming languages, operating systems, and databases – has attempted to address database security using different sets of techniques. The following sections trace the development of these systems and explain how they relate to IFDB.

10.2.1 Languages

Denning proposes language semantics based on the centralized model, which she was the first to formalize [31, 34]. In a system she describes, labels – attached to objects, memory locations, and the program counter – can be checked at runtime. She also notes that information flow properties can be checked statically [31, 36], an idea further developed by Volpano [144] and later realized in Jif.

A renaissance in information flow research accompanied the introduction of the DIFC model in Jif [112, 113] (previously known as JFlow). Jif primarily uses static techniques; it expresses security policies through the type system. It supports limited kinds of runtime label checks for properties that cannot be verified statically.

Fabric [96] extends Jif to distributed, federated systems. It includes persistent objects and transactions, but the paper does not address the possibility of covert channels due to transaction conflicts (section 6.2) or intentional aborts (section 6.1). Follow-on work [95] explains how referential integrity concerns can be codified in a type system; it supports weaker notions of referential integrity than IFDB to handle the possibility that storage nodes might delete objects or disappear entirely.

SIF [21] is a specialization of Jif for web applications. The authors wrote two database-backed applications in SIF. They use DIFC to limit what parts of the

application can communicate with the DBMS, but the database itself is treated as a black box with a uniform label.

Several other languages, like SIF, treat database security as a program correctness problem for the client applications. UrWeb [19], DIFCA-J [151], RESIN [150], and an unimplemented language by Li and Zdancewic [94] are intended to improve web application security by ensuring that the application conforms to a set of information flow assertions. UrWeb and the Li-Zdancewic proposal elevate queries to first-class status in the language and enforce information flow policies statically, whereas DIFCA-J and RESIN work with existing languages and enforce policies dynamically in the language runtime. RESIN also transparently tracks flows through the database via query rewriting; however, the semantics for complex queries, constraints, transactions, and so forth are not developed.

RESIN can prevent SQL-injection attacks by enforcing the property that queries cannot contain low-integrity (that is, unsanitized) data. In contrast, the PHP-IF and Python-IF systems presented in this dissertation use process-level tracking, which is more efficient, but it can't provide the same assurances about certain attacks such as SQL injection.

The use of process-level tracking in PHP-IF and Python-IF reflects a desire to provide a simple, usable, high-performance system that is capable of addressing a wide array of logical errors in applications, such as releasing the wrong user's information. SQL-injection attacks, while rampant, are in some sense an easier problem to solve; they can be prevented, for instance, by using prepared statements. Importantly, however, the database model presented in this dissertation is amenable to fine-grained information flow tracking in the application, as an alternative to PHP-IF and Python-IF. Instead of using the process label in the Query by Label model, a fine-grained DIFC system would use a label that reflects the contamination of the query and the program counter. The tuples returned by the DBMS would also have that label, reflecting the fact that they were produced in response to the query.

Several avenues of research promise to reduce the performance gap between fine- and coarse-grained DIFC, which might make the fine-grained approach in a system using IFDB more palatable. One approach is to use static analysis to reduce the number of dynamic checks; there is evidence that the overhead can be reduced by about 98% [69, 88]. Another is to implement DIFC with tagged-memory architectures [28, 136], but this approach requires specialized hardware. In yet another approach, Laminar [125] achieves efficient, fine-grained tracking under the assumption that most code doesn't operate on sensitive data, while TaintDroid [44]

provides good performance under the assumption that most CPU time is spent in trusted native methods. Neither assumption appears to be appropriate for the applications studied in this dissertation.

10.2.2 Operating Systems

The idea of tracking information flows at the granularity of processes comes from work in operating systems. Coarse-grained tracking is simple and efficient, because the information flow system only needs to interpose on flows that cross a process boundary and only a single label is needed for the entire process.

ADEPT-50 [146] is the first published system to track information flows across processes and files. ADEPT-50 introduced the high-water mark model: Labels are adjusted automatically to reflect the information that has flowed into the respective file or process. Denning [34] notes that this approach leads to covert channels; nevertheless, subsequent systems such as IX [106] and Asbestos [85] are also vulnerable. To avoid these channels, IFDB and most other recent information flow systems require label changes to be explicit.

Asbestos [85], HiStar [152], and Flume [86] adopt the DIFC model. Like Jif, they support discretionary security, in which principals have the authority to define the policies for their own data. Unlike Jif, however, these systems use a capability-based authority structure, rather than a principal hierarchy.

DStar [153], an extension of HiStar, and Aeolus [18] expand the information flow platform to support distributed computing. IFDB is based directly on the Aeolus model, which is described in chapter 3. Aeolus differs from the other DIFC operating systems in two notable ways. First, since revocation and privilege confinement are difficult in capability systems [145], Aeolus uses an explicit principal hierarchy, like Jif. Second, the Aeolus implementation presumes applications are written in memory-safe languages, such as Java, C#, or PHP. Therefore, the Aeolus design includes language features, such as authority closures and shared state, which make it possible to efficiently confine authority and share sensitive data within a process. For example, consider the problem of authenticating users of a website without allowing bugs in the bulk of the application code to compromise users' passwords. An application written in Flume can address this problem by communicating via IPC with a separate, trusted password-checking process. Applications written in the PHP-IF language introduced in this dissertation simply call a trusted authority closure to check the password.

These systems primarily use file systems to store persistent data. Several papers identify information flow control for databases as an area of interest, but do not develop the concept. The Asbestos [85] operating system includes the ability to store labeled data in a database; however, complex queries, transactions, and covert channels are not explored. The authors of HiStar [152] hint at storing labeled data in a database that has a restricted query model, but do not elaborate. This dissertation addresses the needs identified in these papers by developing a comprehensive model for DIFC in databases.

10.2.3 Databases

Prior work on information flow in databases is limited to the centralized model. These systems are called Multi-Level Secure (MLS) databases; most of them were designed to suit military needs, and their models are inspired by the Bell-LaPadula security rules. IFDB differs philosophically from most MLS databases in several respects:

— IFDB is based on discretionary information flow policies, in which data belong to principals, who shape the policy for their own data. In contrast, most MLS databases are based on mandatory policies, defined by an administrator, that constrain all users.

— In order to support discretionary security and allow users to protect their data from each other, IFDB supports a virtually limitless number of tags. MLS databases are typically designed around the assumption that all data can be classified into a small number of security levels and categories.

— IFDB uses information flow as part of a broader methodology of using IFC to build secure software; thus, it includes support for secure programming through ideas such as declassification and authority closures. Earlier database work, in contrast, tends to regard access via different classification levels as a feature to present to users directly. In other words, a user logs in at a particular level, such as *top secret*, and manually issues queries against the database.

Early Models: 1975–1982

A major concern in many of the early systems is how to implement a system with verifiable security, for instance, by minimizing the size of the trusted computing

base. The points of comparison in this section are the security models, however; Section 10.5 describes implementation techniques for secure databases.

The first published MLS database proposal is the Hinke-Schaefer model [68]. The model labels each field with a security level, but several restrictions on labels are needed. Within a tuple, all of the fields that comprise the primary key for a relation are required to have the same level to preserve entity integrity (see section 5.1). Furthermore, other fields are required to have levels at least as high as the level of the primary key.

The I. P. Sharp model [62] classifies data at the level of entire relations, and includes support for both secrecy and integrity. Each user can read relations whose classification is no more restrictive than the user's current secrecy and integrity levels. Unlike IFDB, the I. P. Sharp model allows writes to data that the process is not allowed to see. The authors later observed that to avoid covert channels, the system could not inform the user whether the write succeeded or not [81]. Section 4.3.5 of this dissertation argues against allowing these kinds of writes.

Other granularities of labeling were explored as well. Down and Popek [41] propose a model with field-level classification, while the Naval Surveillance Model [60] allowed data to be labeled at many levels (fields, tuples, tables, or entire databases). Neither paper develops the semantics for relational queries.

The Air Force Summer Study and Subsequent Systems

In the summer of 1982, the US Air Force commissioned a study in Woods Hole, Massachusetts to make recommendations on the design of MLS databases [2]. Many systems were developed as a result of this study [38, 43, 54, 58, 105, 148]. Three of the most influential systems, TRW's MLDS system [54], SeaView [38, 101], and LOCK Data Views [43], are described in this section.

The MLDS system [54] developed at TRW adopts a simple model, using row-level classification like IFDB. Users log in at a given security level, and the system enforces restrictions similar to IFDB's rules for reads and writes (rules 4.1 and 4.2); however, users cannot change their labels or write at any other levels for the duration of the session. The authors briefly identify several important issues such as polyinstantiation (see section 10.4.1), but didn't develop the concepts; instead, the focus is on implementation techniques. (TRW later developed a second system called ASD_Views [55], which has only one label per relation and used views for security; hence, it does not require polyinstantiation.)

Polyinstantiation was later formalized by Denning and Lunt [38] as part of the SeaView project [101]. SeaView supports field-granularity classification as well as relations whose existence is classified, which leads to several different kinds of polyinstantiation:

— *Fields* are polyinstantiated when there are two different values for the field at different secrecy levels, but with the same primary key. This type of polyinstantiation is particularly problematic; it is discussed in depth in section 10.4.

— *Tuples* are polyinstantiated when the primary keys of two tuples have the same value but different secrecy levels. (As in the Hinke-Schaefer model [68], all fields that make up the primary key must have the same secrecy.) IFDB uses only this kind of polyinstantiation: in IFDB, all fields in a tuple share the same label pair.

— *Relations* can be polyinstantiated, so for example, two independent relations with the same name but different secrecy levels (and possibly different schemas as well) can exist. IFDB considers the database schema public and does not support polyinstantiated relations; see section 4.2.

SeaView also has views, but no declassifying or endorsing views. In fact, like other systems based on the centralized model, SeaView lacks declassification and endorsement entirely; instead, it supports *trusted subjects*, which are principals that can read and write within a range of levels, even in violation of the information flow rules. IFDB, in contrast, prevents accidental leaks by disallowing these unsafe flows unless they are explicitly vouched for through an authorized declassification or endorsement operation.

SeaView also introduces the idea of classification constraints (see section 5.5), and the concept is pushed further by the LOCK Data Views (LDV) database system [40, 43, 133]. LDV supports classifying data not only at the granularity of fields, tuples, and relations, but also at the granularity of columns and aggregate types. For example, it permits policies that specify that all salary fields are secret, or that (name, salary) pairs are more sensitive than names or salaries alone.

Commercial and Open-Source Products

Several MLS database products were built following the research prototypes of the late 1980s and early 1990s. These include Sybase Secure SQL Server [138], Trusted

Oracle [13], Informix-OnLine/Secure [72], and Trusted Rubix [73]. See Burns and Koh [15] for a comparison of the systems. All of these products use tuple-level classification and support polyinstantiation, but typically there is an option to disable polyinstantiation. Except for Trusted Oracle, there is no enforcement of referential integrity across security levels. These systems aren't adequate for DIFC; for instance, Sybase Secure SQL Server supports only 16 levels and 64 categories. They use the centralized model and are narrowly tailored to the needs of the military. Since they addressed the needs of a niche market, all of them except for Trusted Rubix were discontinued after several years.

Newer projects have emerged, which extend the security model, but dispense with polyinstantiation and allow covert channels instead. Informix-OnLine/Secure and Trusted Oracle were replaced by Label-Based Access Control [14] and Oracle Label Security [76], respectively. SE-PostgreSQL [82] adds support for SELinux [99] mandatory security policies to PostgreSQL. These systems use extensible per-tuple security labels and user-defined security policies, which can enforce multi-level security, access control lists, or other kinds of policies.

Since these systems support such general security policies, parts of IFDB could have been built on top of one of them – for instance, SE-PostgreSQL. However, the aims are sufficiently different that starting with an ordinary DBMS made more sense. Achieving reasonable performance in SE-PostgreSQL requires tuples to fit into a small number of protection domains, while in IFDB tuples can have many different labels. Also, the Query by Label semantics for integrity constraints and transactions would need to be implemented by modifying the DBMS regardless.

10.3 Fine-Grained Access Control for Databases

In addition to the work on information flow control, there has been research on databases that protect privacy via fine-grained access control. Since these approaches are based on access control rather than IFC, they generally do not address problems with covert channels and updates, nor do they allow for untrusted computations that run on sensitive data without being able to release it. However, like IFDB, they do enable database security policies that protect users from each other.

LeFevre et al. [92] built a system that uses query rewriting to enforce an access control policy written in a policy meta-language. Another technique, authorization views [123], enhances the well-known idea of enforcing access control through views.

Authorization view definitions can be parameterized on the user, much like Query by Label restricts queries based on information flow label. Rather than limiting the results of queries like IFDB does, their system limits the set of legal queries. A query over a set of base relations and views is legal if it is equivalent to another query defined only over authorization views.

10.4 Constraints and Information Flow Control

As discussed in chapter 5, uniqueness constraints, referential integrity constraints, and other types of constraints (handled by triggers in IFDB) can lead to covert channels. There has not been much work on triggers and information flow, but there has been research on referential integrity and on polyinstantiation, the approach to uniqueness constraints that IFDB adopts.

10.4.1 Polyinstantiation

This dissertation regards polyinstantiation as an unfortunate necessity. IFDB uses it only when necessary to prevent covert channels; polyinstantiation occurs when a transaction attempts to insert a tuple that conflicts with a tuple it can't see. Designers of previous systems have taken a different position, motivated by military needs. To illustrate their motivation, suppose a spy conducts both unclassified and secret missions. A user without secret clearance could thus infer that the spy is on a secret mission if the user is unable to see the mission. Thus, it is argued, there is a need for *cover stories* that conceal the existence of the secret mission. However, there is a lack of agreement about basic issues, such as what real-world meaning different combinations of polyinstantiated tuples and fields should have [75, 100].

A series of tables in figure 10-2 illustrate some of the inherent complexity in systems such as SeaView, which support this kind of polyinstantiation at field granularity. Figure 10-2(a) shows the initial state of a SeaView table that lists spies, their locations, and their missions. It has a public entry for Maxwell Smart, in which each field is unclassified. Subsequently, if Maxwell Smart receives a secret espionage mission, SeaView polyinstantiates the tuple, as shown in figure 10-2(b). However, if he now travels to San Monique, and this fact is secret, then SeaView manufactures *two* new tuples, as shown in figure 10-2(c). It's easy to see that this behavior could lead to an exponential number of polyinstantiated tuples!

Spies

agent		location		mission	
unclassified	Maxwell Smart	*unclassified*	Washington	*unclassified*	training

(a) A table in SeaView has separate classification levels for each field.

Spies

agent		location		mission	
unclassified	Maxwell Smart	*unclassified*	Washington	*unclassified*	training
unclassified	Maxwell Smart	*unclassified*	Washington	*secret*	espionage

(b) If a user updates Maxwell Smart's mission to the secret value espionage, SeaView manufactures a tuple to handle the polyinstantiated field.

Spies

agent		location		mission	
unclassified	Maxwell Smart	*unclassified*	Washington	*unclassified*	training
unclassified	Maxwell Smart	*unclassified*	Washington	*secret*	espionage
unclassified	Maxwell Smart	*secret*	San Monique	*unclassified*	training
unclassified	Maxwell Smart	*secret*	San Monique	*secret*	espionage

(c) If the user subsequently updates Maxwell Smart's location to the secret value San Monique, SeaView manufactures *two* additional tuples to encode the possible combinations of unclassified and secret polyinstantiated fields.

Spies

agent		location		mission	
unclassified	Maxwell Smart	*unclassified*	Washington	*unclassified*	training
unclassified	Maxwell Smart	*unclassified*	Washington	*secret*	espionage
unclassified	Maxwell Smart	*secret*	San Monique	*secret*	espionage

(d) SeaView does not allow tables such as this one. Jajodia and Sandhu [75] argue that it should be supported.

Figure 10-2: These tables illustrate how field-level polyinstantiation can lead to an explosion in the number of tuples. In each of the tables shown, the primary key is agent.

SpyLocations

_label	agent	location
{}	Maxwell Smart	Washington

SpyMissions

_label	agent	mission
{}	Maxwell Smart	training
{secret}	Maxwell Smart	espionage

Figure 10-3: IFDB can support a labeling scheme equivalent to the one shown in figure 10-2(b) by defining Spies as a view: $Spies = SpyLocations \times SpyMissions$.

An intuitive justification for SeaView's choice is that there are just two polyinstantiated fields, location and mission, and the tuples just represent all the combinations of those fields. This means that Maxwell Smart can only have a single value for a given field at each classification level. Thus, if his unclassified location changes from Washington to New York, the first two tuples in figure 10-2(c) inherit the change.

Jajodia and Sandhu [75] argue that tables such as the one shown in figure 10-2(d) should be allowed. But this proposal raises new questions. What does the table even mean? If a process later changes Maxwell Smart's location at the unclassified level, does the update affect the first tuple, the second, or both? Attempts to address this problem include BELIEVED BY clauses in the Smith-Winslett model [131] and data-borrow integrity in the MLR model [128].

Since IFDB supports labeling and polyinstantiation only at the granularity of tuples, these problems do not arise. However, suppose that it is desirable to give Maxwell Smart's location and mission different labels. Section 4.5.4 explains how to do this by decomposing the table into two tables as shown in figure 10-3. Now Spies can be defined as a join over the two tables.

Note, however, that this approach has the same problem as SeaView: if a secret location for Maxwell Smart were inserted into SpyLocations, the resulting Spies view would include four tuples. In fact, SeaView's implementation of field-level classification uses a similar decomposition internally. As described in section 5.2.2, however, IFDB regards polyinstantiation as a consequence of mistakes, which are expected to be rare. In fact, adding a secret location for Maxwell Smart to SpyLocations would not be allowed; the process running with the secrecy label {secret} would see the existing tuple and abort with a primary key constraint violation.

Many MLS database systems do support this type of polyinstantiation, where a process is allowed to insert a tuple that conflicts with a lower-labeled tuple. Two types of polyinstantiation are recognized:

— *Invisible polyinstantiation* occurs when a process inserts a tuple that conflicts with a higher-labeled tuple, which it is not allowed to see. This is the type of polyinstantiation IFDB supports, and it is needed to avoid covert channels.

— *Visible polyinstantiation* occurs when a process inserts a tuple that conflicts with a lower-labeled tuple, which it *can* see. This type of polyinstantiation does not prevent any covert channels, and IFDB does not allow it. Prior research has justified it through the perceived need for cover stories, and through the idea that a user with a high secrecy level should not be inconvenienced by a conflicting tuple at a lower level. The latter argument confounds secrecy and integrity.

Supporting both types of polyinstantiation is equivalent to including the label in every primary key, and in fact, this is how Trusted Oracle implements it. Sybase Secure SQL Server and Informix-OnLine/Secure make both types of polyinstantiation mandatory. Trusted Rubix has per-table configuration settings to make each type of polyinstantiation optional.

10.4.2 Referential Integrity

Tuples with different labels that are governed by a referential integrity constraint can create covert channels. IFDB's approach to referential integrity, explained in section 5.3, is to require that tuples with foreign keys are properly vouched for via declassification or endorsement when they are inserted. Thus, the inserting process acknowledges the fact that the related tuples reveal information about each other.

IFDB appears to be the first system to take this approach. Most prior work has focused on the problem of referential ambiguity [91, 128, 129], which is actually a consequence of polyinstantiation! Approaches to avoiding covert channels due to referential integrity constraints take two forms:

— SeaView [101], Trusted Rubix [73], Informix-OnLine/Secure [72], and the MLR model [128] require the referring and referenced tuple to have the same security level. This approach is too limiting to enforce many reasonable constraints. For instance, it does not allow for enforcement of the constraint that every user has a password, assuming that one table stores user account information, a separate table stores passwords, and tuples in the two tables have different labels.

— Trusted Oracle [13] and Sybase Secure SQL Server [138] follow a proposal by Gajnak [52] to require the referring tuple to have the same security level or higher. Attempting to delete the referenced tuple introduces a channel. Proposed solutions include ignoring the problem, logging the fact that a leak occurred, or using access control to restrict deletion. With the last option, the implicit leak still occurs, but it is limited to authorized users. IFDB did not adopt such an approach because it violates the principle that all unsafe information flows should be explicit.

Millen and Lunt [109] develop a model for object-oriented databases that handles referential integrity by drawing a distinction between the classification of an object and the classification of the *existence* of the object. Adopting this view in IFDB might permit some of the referential integrity rules to be relaxed, but having two kinds of classifications for each tuple complicates the model.

10.5 Secure DBMS Architectures

Since IFDB is implemented as a modified version of the PostgreSQL DBMS, it has a large trusted computing base (see section 2.3). This approach is common: MLDS [54], ASD_Views [55], and virtually all commercial MLS databases [13, 72, 73, 138] are built this way. The size of the TCB is of some concern, however; in fact, twenty-three vulnerabilities in PostgreSQL that might lead to data leaks were reported in the five-year period from 2007 to 2011 [114]. Two major approaches to securing database systems have been proposed. This section examines each one, and discusses its applicability to IFDB.

10.5.1 The Kernelized Approach

One approach is to rely on lower-level protections, such as those provided by the operating system, to ensure data security. Hinke and Schaefer [68] first proposed this idea, which is referred to as the Hinke-Schaefer (or *kernelized*) architecture. In their design, each relation is partitioned vertically and horizontally into fragments so that all the information in each fragment has the same security level, and the fragments are then stored in separate Multics files. The underlying operating system then enforces the information flow rules.

Subsequently, systems such as SeaView [101] and the OS-MAC version of Trusted Oracle [13, 143] were built on this idea. SINTRA [49, 50] uses a variant of this approach. Instead of enforcing isolation via a trusted operating system, SINTRA uses physically separated database systems connected via a trusted switch. Each security level has a database, which contains all of the data for that level and below.

These architectures are plagued by poor performance [53, 67], and concurrency control for them is challenging [105]. DIFC systems such as IFDB use many different labels rather than a small set of security levels, which would exacerbate the problem. For instance, in a medical information system, the patient records table would need to be split into a separate file for each patient. Therefore, relying on a DIFC operating system for security does not seem like a practical solution.

10.5.2 Trusted Proxies

An alternative approach is to rely on a trusted proxy to mediate all communication with the database and ensure that the security policy is followed. The proxy is simpler than the DBMS, and therefore easier to verify. But how does the proxy ensure that the database is doing the right thing?

One answer is the integrity-lock architecture, which was proposed independently by Graubart [58] and Denning [35], and subsequently implemented [59]. The idea is to have the proxy compute a message authentication code (MAC) over each datum and its label before storing it in the database. The MAC cryptographically binds the datum to the label, so the DBMS can't change the label without the proxy noticing. Later, when the same datum is read from the database, the proxy verifies the MAC and checks that the label satisfies the information flow rules for the query before returning the results to the application.

Another technique is to encrypt the data before storing it in the database. Until recently, it was not known how to query encrypted data efficiently, and substantial client-side processing was required [63]. However, advances in homomorphic encryption have made it possible to securely compute on encrypted data on the server. CryptDB [119] uses a trusted proxy to encrypt data, and executes most common types of queries using homomorphic encryption, with about 20% overhead. A notable advantage of this technique is that it does not assume that all communication with the database goes through the proxy. A disadvantage is that the DBMS still learns some information about the encrypted data, which is necessary in order to execute certain types of queries efficiently.

Neither the integrity-lock architecture nor encryption ensure integrity of results. This is a particularly bad problem for the integrity-lock approach because a malicious DBMS could leak data indirectly through bogus responses. For instance, in responding to a query with an empty (public) secrecy label, the database might use secret data to decide which public tuples to return. Byzantine-fault-tolerant replication [142] addresses this problem by using multiple databases and trusting the result only if a quorum of them agree.

A proof-of-concept proxy for Query by Label has been built, and is described in section 8.6. Future work could include a complete proxy-based implementation, combined with encryption or Byzantine-fault-tolerant replication, to provide security with minimal trust in the DBMS. Section 11.2.3 explains how an implementation based on replication might work.

10.6 Transactions

This dissertation has identified two covert channels related to transactions. First, section 6.1 notes that aborting a transaction can be used to signal secret information. Second, sections 6.2 and 8.5.3 explain how conflicts among concurrent transactions can also lead to unsafe flows. This section covers prior work on these topics.

10.6.1 Abort Channels

The only prior system to address abort channels is Fabric [96], which handles the problem differently from IFDB. Whereas IFDB requires that a process have a sufficiently uncontaminated label in order to commit a transaction, Fabric enforces an analogous restriction at all the points where a transaction might *abort*. Specifically, a Fabric transaction cannot abort while the label of the process is higher than at the start of the transaction.

Fabric is based on the Jif language, so it uses static analysis to determine all the points where an abort might occur. These points include any explicit abort statements, as well as any code that might throw an unchecked exception. Unfortunately, due to fundamental limitations of static analysis, such points are pervasive [80]: dereferencing an object creates the potential for a null-pointer exception, and accessing an array may lead to an array-bounds exception, for instance. In fact, any read or write of an object that can be accessed by another transaction could cause an abort

due to a concurrency conflict, so it's not clear how transactions restricted by Fabric's rule could get any useful work done while running with a higher label than it started with. IFDB avoids these problems, as well as the requirement of statically analyzing transactions in advance, by restricting commits instead of aborts.

10.6.2 Secure Transaction Scheduling

Section 6.2 describes how IFDB ensures that transaction conflicts do not introduce signaling channels, and appendix A describes an alternative for snapshot isolation. This problem has been studied extensively in the context of MLS database systems, but none of the prior systems allow processes to change labels in the middle of a transaction.

The first system to address the problem is the OS-MAC version of Trusted Oracle. In Trusted Oracle, read/write transactions use strict two-phase locking for writes at their own security level, and a variant of multiversion timestamp ordering (MVTO) [122] to read data at their own level and lower [103, 143]. MVTO provides snapshot isolation [10], which is weaker than serializability; the IFDB prototype provides only snapshot isolation as well, although the IFDB model supports serializability. Trusted Oracle and a number of subsequent proposals [4, 74, 116] use a restricted model where all of the writes for a transaction must have the same label.

Keefe and Tsai [79] present a transaction scheduler that provides serializability. Their approach also restricts transactions to have a single label, but transactions are allowed to do blind writes of higher-labeled data. The algorithm is also based on MVTO, modified so that the read timestamp assigned to a transaction precedes the timestamps of any concurrent transactions at lower levels. Unfortunately, in their protocol, long-lived transactions at low levels can cause higher-level transactions to run far in the past, and can cause starvation. Trusted Rubix appears to use their algorithm, or a related one, but provides the option of relaxed consistency to avoid these problems. In IFDB, there is an additional problem, though: If a client executes a transaction, raises its label, and executes another transaction, the second transaction might be serialized before the first. While this behavior doesn't violate serializability, it does violate sequential consistency and may lead to confusion.

IFDB allows processes to change their labels in the middle of a transaction, so that tuples with different labels can be written atomically, as described in section 4.3.5. IFDB's transaction clearance approach described in section 6.2 is a simple and practical way to prevent label changes from introducing unauthorized channels.

However, appendix A presents alternative semantics based on flow-safe scheduling, for which conflicts are more problematic. Since transactions can conflict with any other transaction within the range of labels they have written at, flow-safe scheduling sometimes requires them to be aborted. These aborts can lead to starvation. Prior work has looked at this problem and proposed various solutions, such as reduced atomicity guarantees [12], accepting some covert channels [104], or assuming one-shot transactions that can be analyzed in advance [132]. In addition, schedulers that provide various forms of relaxed consistency to avoid the starvation problem have been proposed [74, 103, 117]. Other work has studied the problem in a context where all the data with a given label is stored in its own database [5, 26, 77, 105], but as explained in section 10.5.1, these architectures aren't amenable to DIFC. There has also been work on secure two-phase commit protocols [121].

While this work addresses conflicts among transactions that access the same data, it does not address timing channels that may arise as a result of contention for other shared resources, such as the disk, memory, and CPU. Timing channels cannot be avoided entirely, except by preallocating these resources and forbidding dynamic sharing across security domains – an impractical limitation. However, timing channels can be mitigated [6, 78, 97], for example, by quantizing response times of queries.

10.7 Inference and Statistical Privacy

Many systems that process data with different secrecy labels are vulnerable to the *inference problem*: It is possible to learn something about secret data by looking at less secret data. For example, the issues with foreign keys covered in section 5.3 are instances of the inference problem. In systems based on access control, and in MLS database systems that have trusted subjects, it is easy to create inference channels inadvertently. The Query by Label model, however, ensures that if data are correctly labeled in the first place, the *only* way to create such a channel is through explicit use of authority (that is, declassification or endorsement).

Declassification and endorsement provide a simple way to reason about channels that might leak information. However, the programmer must still make decisions about what releases are safe. Therefore, it is important to have a framework for understanding how much information is being released, for instance, by "anonymized" records and statistics. Statistical databases and anonymization have been widely

studied [33, 102, 137]. Recently, the introduction of Differential Privacy [42] has put the field on strong theoretical grounds. Unfortunately, there is a tradeoff between the amount of information that is released and the probability that an attacker can infer sensitive information. Several classes of queries that protect differential privacy have been studied [16, 65]. Furthermore, techniques have been developed to bound the information leaked over the course of many queries [107]; future research challenges include incorporating such systems into IFDB, and generalizing the types of queries that are possible. Section 11.2.2 suggests some specific directions.

Chapter 11 Conclusions

This dissertation has presented IFDB, the first system to provide security for databases by using decentralized information flow control (DIFC). IFDB fills an unmet need by extending the recent DIFC work that has been done in the context of operating systems and programming languages to support persistence through relational databases. It also incorporates some techniques from multi-level secure databases, such as label constraints and polyinstantiation; however, it substantially extends that work with new abstractions for secure information flow, a decentralized information flow model, support for foreign key constraints, the ability to change labels in the middle of a transaction, and techniques for avoiding a variety of covert channels. This chapter summarizes the main contributions of this thesis and identifies opportunities for future work.

11.1 Contributions

This thesis has advanced several arguments in support of the claim that DIFC is a good way to secure databases containing sensitive data. The main advantages of using the IFDB model are:

— *Flexibility.* Since IFDB tracks information flows between the application and the database, it makes it possible to perform computations on sensitive data in both the database and the application, without the need to trust the code. In

contrast, database access control systems cannot control how information is used once it has been read by an application.

— *End-to-end security.* Information flow policies label sensitive data when they enter the system and control what can be released. Much of the code involved in intermediate processing of the data does not need to be trusted, and having less trusted code improves security.

— *Simplified reasoning about security.* Reasoning about data privacy has been a continual challenge; privacy requirements can be complex, and often revealing one piece of information reveals something more sensitive by implication. IFDB's principle of explicit release helps programmers understand the implications of their actions by requiring each action that has the potential to leak information to be explicitly vouched for through declassification.

— *Decentralized policies.* Unlike the earlier work on multi-level-secure database systems, IFDB supports the DIFC model. Thus, it is suitable for systems with many different users, where it is important to protect each user's data from other users. It also allows sharing and discretionary security, so each user can define a policy for her own data.

In DIFC systems, processes are contaminated by what they read, and the information flow policy restricts what they can do based on their contamination. IFDB introduces the Query by Label model as a practical way for database-backed applications to control their contamination. Additionally, IFDB adopts the Aeolus information flow model, which includes a number of abstractions to help processes manage information flows and authority. This work extends the model to handle computations in the database by adding new abstractions, such as declassifying views, endorsing views, and stored authority closures. Furthermore, IFDB provides a uniform API on the application side and the database side, so developers can put computations in the applications or in the DBMS – whichever is most appropriate for the task at hand.

Past work on IFC in database systems has argued that it is necessary to allow different fields within a tuple to have different information flow labels, but these fine-grained labels come at significant expense. This dissertation has shown that tuple-level labels are sufficient when combined with declassifying and endorsing views. These kinds of views make it possible to achieve data independence for labeled

tuples, which ensures that programmers are not forced to arrange data according to security concerns.

Another contribution of this dissertation is the development of appropriate semantics for constraints to ensure that they do not create a means to circumvent the information flow policy via covert channels. For uniqueness constraints, IFDB adopts a simplified version of the polyinstantiation concept from SeaView [101]. Since polyinstantiation is regarded as an unfortunate necessity, chapter 5 also introduces a way to use label constraints to avoid polyinstantiation. IFDB also introduces DECLASSIFYING and ENDORSING clauses to manage information disclosures via foreign key constraints involving tuples with different labels. For other types of constraints, section 5.4 develops a methodology for understanding the information flows that can happen through the actions of user-defined triggers.

Transactions are another important DBMS feature that introduce new challenges in information flow systems, for two reasons. First, applications can choose to abort their transactions after reading secret information, which may leak information about the secret. Second, conflicts among transactions at different secrecy levels can lead to covert channels. Prior work has sidestepped this problem, either by restricting what may be written in a single transaction [4, 13, 74, 116] or by providing weaker semantics, such as blind writes [79]. In chapter 6, this dissertation introduces new ways of addressing the problems; additionally, section 8.5.3 describes the impact of IFC on the implementation of concurrency control in the DBMS, and appendix A introduces alternative semantics that are appropriate for snapshot isolation.

Chapters 7 and 9 demonstrate that IFDB is practical by evaluating a prototype implementation, along with a new DIFC application environment, PHP-IF. Chapter 7 shows that IFDB is easy for developers to use, and that it improves the security of real applications. Two applications were studied, both of which have complex security policies, and both of which had security bugs. It was possible to convert them in a straightforward way by defining a few kinds of principals, tags, and authorizations; making decisions about how tuples of various tables should be labeled; and then defining some authority closures to enforce policies not directly captured by the labels. The result in both studies was an implementation that prevented a number of security bugs. In each case there was a limited amount of code that affected security; only this code needs to be verified to ensure that the policies are implemented properly. Chapter 9 completes the evaluation by showing that IFDB's performance for real workloads is nearly as good as in the original PostgreSQL implementation, upon which the IFDB prototype is based.

11.2 Future Work

IFDB is the first work to add decentralized information flow control to a relational database system, and as such, it opens up new possibilities for future research on DIFC for databases. There are three areas that appear to have interesting prospects. First, theoretical work could formalize the information flow model and prove that it is correct. Second, some applications could be simplified through extensions to the model. Third, the IFDB implementation could be refined to improve its security and performance.

11.2.1 Proofs of Noninterference

The semantics of IFDB are carefully constructed to prevent direct and indirect information leaks, and this dissertation has explained informally why the model is correct. A greater degree of assurance could be achieved by formally modeling the system and constructing proofs of correctness. Noninterference [57] is a commonly accepted definition of what it means to be free from covert channels, so proofs of noninterference for the model would be a good contribution.

A process P_1 is noninterfering with another process P_2 if the execution of P_2 is unaffected by P_1. In other words, P_2 should behave the same as if P_1 did not exist. For information flow systems, noninterference is typically used to establish that a process with a high label pair is noninterfering with any lower-labeled process.[1] This definition precludes the possibility that the high process could leak something to the low process via a covert channel.

In the context of IFDB, processes communicate via the database. (The Aeolus model allows direct communication as well, but that is a separate concern.) The goal is to prove that high-labeled processes do not affect lower-labeled processes through their interactions with the DBMS. It is useful to think of the database as having two parts: a high part and a low part. Then the proof can be broken down into the task of establishing the following two properties:

1. A high process does not affect the low part of the database.

1. In general, there are many processes with many different labels, but it suffices to think of noninterfering pairs of processes, one of which is the "high" process and the other the "low" process. The proof can then be extended to arbitrary mixes of processes via induction. Also, if two processes have incomparable labels, such as {alice_medical} and {bob_medical}, then they should both be noninterfering with each other.

2. The results seen by a low process depend only on the low part of the database. Furthermore, the effect of the low process on the low part of the database is the same regardless of the contents of the high part of the database.

To handle concurrency conflicts, it is important that the formal model specify how high transactions could affect lower transactions through the concurrency control mechanism. Additionally, declassification complicates matters because by its very nature, it allows violations of noninterference in a controlled way. Li and Zdancewic [93] describe one way to model declassification.

Thinking about information channels in terms of the high and low processes interacting with different parts of the database was helpful in developing the semantics for constraints and transactions. Formal noninterference proofs might offer further insights and provide additional assurance that the semantics are correct.

11.2.2 Extensions to the Model

IFDB presents a simple but powerful model for applications, and the case studies presented in chapter 7 demonstrate that the label model and abstractions such as authority closures and declassifying views provide all the expressive power applications need. In fact, any security policy can be expressed through an authority closure. However, extensions to the model could increase convenience and security for certain types of applications by allowing them to express certain information flow policies more directly. This section describes some extensions that seem useful.

Weaker Confinement Rules for Integrity

Section 11.2.1 explains that IFDB's information flow model uses noninterference as a basis for reasoning about covert channels. Noninterference is a strong definition, which disallows all possible ways that a process with a high label might communicate with a process with a lower label. A weaker requirement would make some of IFDB's rules less restrictive, which would be more convenient for application developers, and in some cases, reduce the amount of trust that must be placed in code.

Chapter 5 illustrates that permitting additional channels could lead to harmful leaks for secrecy, but for integrity labels, the consequences are less harmful. For constraints and transactions, for instance, using traditional database techniques instead of the rules described in chapters 5 and 6 would allow low-integrity processes

to launch denial-of-service attacks against high-integrity processes. However, low-integrity processes could not corrupt high-integrity processes with low-integrity data. Convenience and reduced trust are the rewards for accepting the possibility of denial-of-service attacks: processes would no longer need to use authority to vouch for inserts every time an inserted tuple refers to a higher-integrity tuple via a foreign-key constraint.

In fact, although IFDB provides uniform and strict rules by default, it does allow programmers to create weaker rules. Programmers must implement these weaker rules themselves – for instance, by making a rewrite rule that converts inserts into a table into a call to an authority closure that handles the necessary endorsements. Implementing such rules and closures could be tedious, however, so it would be more convenient to have a more direct mechanism. More experience with integrity is needed in order to identify common patterns, so that they can be simplified with the right abstractions.

Abstractions for Data-Parallel Computation

IFDB makes it possible to produce derived data on-the-fly through the use of triggers. However, some processing needs are better served through batch jobs – for instance, producing monthly patient bills from medical records. A challenge with these batch jobs is how to process information with a variety of different labels while still ensuring secure information flow. When producing patient medical bills, for instance, it is important that Alice's medical records do not wind up in Bob's bill.

One could imagine handling this kind of computation in the database with a query such as the following:

```
INSERT INTO PatientBilling
    SELECT compute_bill(patient_name, array_agg(proc_code))
        FROM PatientVisits
        WHERE date_trunc('month', visit_date) = lastmonth
        GROUP BY patient_name
```

This query groups all of the patient visit records for a given month and sends them to the compute_bill stored procedure, which computes the bill for the patient. However, the process would need to run with the all_patients_medical tag in its secrecy label in order to read all the patient records in a single query, and thus each billing record would be contaminated with the all_patients_medical tag. Of course, there are ways around this limitation, such as having the compute_bill procedure run with high

authority so that it can declassify all_patients_medical and insert the billing tuple with the correct label. Nevertheless, it would be more straightforward if there were a way to arrange so that each invocation of compute_bill ran with the labels of the corresponding patient tuples, rather than secrecy label {all_patients_medical}.

It would also be useful to have a mechanism to perform processing such as the medical-to-billing computation in the application. In the Query by Label model, such computations require more contamination than they ought to: The query results have a single label pair, which is the label pair of the process when it issued the query. To achieve good performance, it is imperative that the process fetch multiple patients' records in a single query, but this means that the process secrecy label must be {all_patients_medical}. Ideally, the system would support an iterator abstraction that makes it possible to iterate over query results, processing each tuple with only the contamination associated with that tuple.

The challenge in supporting these kinds of extensions is that they can lead to covert channels. For instance, any errors that occur in producing a single patient's bill cause the entire bill computation transaction to fail, and other patients will notice the failure when they do not receive their bills. New abstractions might solve the problem by splitting the computation into multiple transactions or recording the failures in a log instead of aborting.

Integration with Fine-Grained DIFC Systems

Coarse-grained DIFC systems like Aeolus, upon which IFDB is based, use a single label to track the contamination of an entire process. The advantage of this approach is that it has good performance without requiring static analysis. However, if a single process handles many patients' medical records at the same time, the system will not be able to track flows of Alice's medical data and Bob's medical data independently. For web applications such as the CarTel and HotCRP applications discussed in chapter 7, this isn't a problematic limitation, since web applications typically process data on behalf of a single user at a time.

However, data-parallel tasks such as the ones described in the preceding section might benefit from an information flow system that tracks flows at the granularity of bytes or objects [112, 113, 125, 150, 151]. Therefore, it is worthwhile to investigate how Query by Label can be adapted to work with these systems.

Dynamic Tag Groups

Section 7.3.4 discusses how experiences with HotCRP motivated the need for dynamic tag groups. Dynamic tag groups extend compound tags by providing efficient ways to refer to sets such as the set of all paper review tags that program-committee member Bob is not conflicted with. However, dynamic tag groups can also create new information channels, because changes in the tag group affect the contamination of processes and tuples that already exist. Therefore, additional research is needed to work out appropriate semantics for dynamic tag groups.

Abstractions for Statistical Databases

Declassification is a tool that allows programmers to express information flow policies, but the onus is still on the programmer to determine when a particular release of information is appropriate. In many cases, such as releasing Alice's medical record's to Alice's doctor, it is quite clear that the release should be allowed. However, for statistical information, the answer is less clear. Does the average age of all the HIV patients in the clinic reveal too much sensitive information about the patients? If the clinic is very small or the query is asked frequently enough, it might.

There is a rich theory on quantifying the impact of releasing statistical data, as outlined in section 10.7, and there are many techniques for minimizing the information leaked. The question is whether new abstractions would make it easier to ensure that declassified statistical data does not leak too much information. For example, declassifying views that produce statistical data might require the ability to limit the rate of queries or the total number of queries over the view, so that an attacker cannot learn too much information by querying the view repeatedly. Additionally, statistical queries need to be constrained so that they are always executed over a sufficiently large population, so that little personally identifiable information is revealed; thus, it is important to provide a mechanism to enforce such constraints. Other extensions that might be useful for statistical computations include aggregation constraints [133], which specify that a combination of values has a higher secrecy label than the union of the labels of the individual values, and aggregate operators that support statistical techniques for protecting privacy, such as data swapping [27].

11.2.3 Extensions to the Implementation

Chapter 9 shows that the IFDB prototype performs quite well. Therefore, with the exception of possibly adding support for indexes containing labels (see section 8.1.1), compelling extensions to the model aim to improve security. This section proposes two such extensions.

Reducing the Trusted Base

IFDB has a large trusted computing base, so there is a real concern that bugs in the DBMS could compromise security. Typically the database is behind a firewall, and the applications are the weakest link in security. Nevertheless, it is important that the database system be trustworthy as well. As section 10.5 explains, there has been some work on database architectures with better security. However, many of the proposals, such as storing differently labeled data in different files, are not practical in a DIFC system.

One practical technique to reduce the amount of trust in the DBMS is to use a trusted proxy and Byzantine-fault-tolerant replication [142]. The proxy receives queries from applications and forwards them to several independent database systems. The database systems are not connected to the Internet and can communicate only via the proxy. The proxy is responsible for verifying that the responses returned to applications are correct according to the Query by Label model. Unfortunately, the verifier can't just look at the labels of the tuples to determine if the results are okay, because a buggy database might return public tuples, but decide *which* public tuples to return based on secret data. Instead, the proxy accepts a response if a majority of the databases agree. Thus, if there are three databases, an attacker would have to find a similar bug in two of them (or in the proxy, which is simple and presumably more likely to be correct) in order to break the system.

To avoid correlated failures, three different databases could be used: for instance, PostgreSQL, Oracle, and MySQL. Since it would be prohibitive to add support for the IFDB model to three different database implementations, the model could instead be implemented in the proxy via query rewriting. A preliminary query-rewriting implementation of Query by Label is described in section 8.6.

Covert Channels in the Implementation

The IFDB model is designed to prevent covert channels in the abstract model. However, section 8.5.1 points out that the prototype implementation is still vulnerable to timing channels. Timing channels rely on the fact that some queries can be answered faster than others, so techniques for mitigating timing channels must delay some responses to limit the amount of sensitive information that could be revealed by timing. This increase in query latency has an impact on performance, and further studies would be needed to assess the real-world performance impact of timing attack mitigation.

Appendix A Flow-Safe Scheduling

Section 6.2 discusses how IFDB handles *conflict channels*, which might allow processes to signal information to each other via concurrency conflicts in transactions. It is important that IFDB does not allow these channels to create unsafe information flows. Section 6.2 proposes two ways to do this: transaction clearance and flow-safe scheduling. This appendix reviews the problem and explains the flow-safe scheduling approach.

Flow-safe scheduling has advantages compared to using transaction clearance. First, it does not add any additional rules that the application must follow. Second, it maintains the desirable property that all unsafe flows be vouched for with an explicit use of authority. The alternative solution using transaction clearance allows unsafe flows that involve *implicit* use of authority, as section 6.2 explains.

However, flow-safe scheduling also has a disadvantage: it is not possible to implement it in a practical way while ensuring serializability. This downside is not a problem for the IFDB prototype, which is based on PostgreSQL 8.4.10, because the highest level of isolation that PostgreSQL 8.4.10 supports is snapshot isolation. (More recently, PostgreSQL 9.1 added support for serializability.) Appendix A.1 reviews the conflict-channel problem and defines the goal. Appendix A.2 explains how to do flow-safe scheduling for snapshot isolation, and appendix A.3 shows why flow-safe scheduling is not practical for any concurrency control scheme that provides serializability and general transactions.

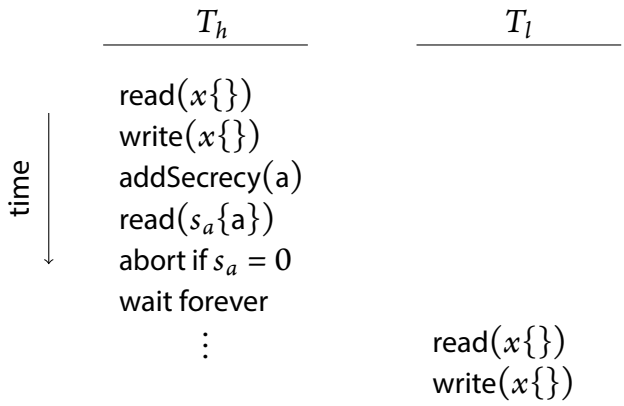


Figure A-1: A conflict between a high-labeled transaction T_h and a low-labeled transaction T_l could potentially introduce a covert channel. T_h and T_l both start with empty labels, and then T_h raises its label. The notation write(x{}) means that the transaction writes tuple x, which has label {}.

A.1 Transaction Conflicts and Conflict Labels

Recall the example first introduced in chapter 6, reproduced in figure A-1, that illustrates how transaction conflicts could be used to signal secret information in violation of the information flow rules. A process running with an empty label starts a transaction T_h and reads and writes a public tuple x. Then the process raises its label and reads a secret. It decides to abort the transaction or wait forever based on the value that it read. Subsequently, another process with an empty label starts a transaction T_l and tries to read and write x. However, T_l will conflict with T_h if T_h did not abort. If the DBMS causes T_l to abort or block due to the conflict, the process that issued T_l will learn something about the secret, even though it is running with an empty label.

Based on the example, a first attempt at a general principle for conflict-channel prevention might be that a transaction T_l should not have to block or abort as a result of the actions of another transaction T_h unless T_h's labels are less restrictive than or equal to T_l's. However, a slightly weaker definition is needed because processes that declassify and endorse within transactions might have conflicts due to operations they executed prior to declassifying or endorsing. For example, suppose a process writes a tuple with label {alice_medical} and declassifies in the middle of a transaction, so that its new label is {}. Until the transaction ends, it may still be

affected by conflicts with other transactions with label {alice_medical} because it wrote a tuple with that label. Since the process just used authority to declassify for alice_medical, the fact that it may subsequently learn one bit of information protected by alice_medical due to a conflict isn't considered significant.

Conflict labels help formalize the idea, taking declassification and endorsement into account. The *secrecy conflict-label* of a transaction T, written $T.L_S^C$, is the union of the associated client's current secrecy label and the set of tags the client has declassified since starting the transaction. Similarly, the *integrity conflict-label* of a transaction T, written $T.L_I^C$, is the intersection of the client's current integrity label and the set of tags the client has endorsed since starting the transaction. Note that the conflict-label pair of a transaction is always at least as restrictive as its current label pair, and the conflict label can never be lowered. The goal of preventing conflict channels can be expressed through the following property, which says that a transaction T_h should not interfere with a transaction T_l unless the label pair of the process that issued T_h is less restrictive than T_l's conflict-label pair.

Property A.1. (Transaction Conflicts)
Suppose transaction T_l has conflict-label pair $(T_l.L_S^C, T_l.L_I^C)$. Let $(T_h.L_S, T_h.L_I)$ denote the commit labels of another transaction T_h, if T_h has committed, and the current label pair of the process that issued T_h otherwise. If $(T_h.L_S, T_h.L_I) \not\preceq (T_l.L_S^C, T_l.L_I^C)$, then T_h must be noninterfering with T_l.

The statement that T_h is noninterfering with T_l means that in any schedule of concurrent transactions, T_l will behave identically regardless of whether T_h is included in the schedule or not. Informally, behaving identically means returning the same results to the client and having the same effect on the database. Goguen and Meseguer [57] introduce and formally define noninterference, and Keefe and Tsai [79] adapt the meaning for transactions specifically.

A.2 Flow-Safe Scheduling for Snapshot Isolation

How can a DBMS satisfy property A.1? To answer this question, it is necessary to look at how conflicts between transactions arise, and how database systems resolve them. In general, transactions such as T_h and T_l from the preceding example can have three types of conflicts:

— *Read-write conflict.* A conflict occurs because T_h reads x and T_l subsequently writes x.

— *Write-read conflict.* T_h and T_l conflict because T_h writes x and T_l subsequently reads x.

— *Write-write conflict.* A write-write conflict occurs when T_h and T_l both write x.

Each conflict between committed transactions represents an edge in a directed graph called the *conflict graph*. An edge from T_1 to T_2 means that T_2 must appear to happen after T_1, either because T_2 wrote a new version of x after T_1 observed the old version (a read-write conflict), because T_2 observed a value that T_1 wrote (a write-read conflict), or because T_1 and T_2 both wrote the same tuple and T_2 wrote it second (a write-write conflict). The job of the transaction scheduler is to prevent cycles in the conflict graph, since a cycle would mean that neither T_1 nor T_2 could be ordered before the other.

There are a three kinds of concurrency control strategies. Pessimistic techniques have transactions acquire locks on data they read and write, so that subsequent transactions will block if they attempt conflicting operations. Optimistic techniques allow the conflicts to occur, but abort any transactions that would produce cycles in the conflict graph. Multiversion techniques keep multiple versions of data; thus, a write-read conflict between T_h and T_l can be avoided by having T_l read the old version of x, before T_h's write.

The general idea in flow-safe scheduling is to either prevent cycles in the conflict graph that involve transactions with different labels, or to resolve conflicts by aborting the transaction with the higher label pair. Prior work described in section 10.6.2 takes the former approach: it uses multiversion concurrency control to ensure that there is never an edge from a low-labeled transaction to a concurrent high-labeled transaction. However, the technique does not work if processes are allowed to change labels in the middle of a transaction. This appendix proposes to use the latter approach to achieve flow-safe scheduling for snapshot isolation.

In snapshot isolation, each transaction executes all of its reads with respect to a consistent snapshot of the database. Snapshot isolation is weaker than serializability. The only types of conflicts that cause the scheduler to abort or block a transaction are write-write conflicts. In IFDB, if transaction T_l writes a tuple, and this operation conflicts with an earlier write to the tuple by transaction T_h, then one of three things is true:

1. T_h is committed. Then by rule 6.1 (page 85), the process that committed T_h must have had a label no more restrictive than the current label of T_l's process.

2. T_h is still active, and the processes that issued T_h and T_l have the same labels.

3. T_h is still active, and the processes that issued T_h and T_l do not have the same labels. Then T_h's process must have had T_l's labels previously, since it wrote a tuple with those labels; however, it subsequently changed its labels.

In the first case where T_h is already committed, it is okay (and necessary) to abort T_l. The other process doomed T_l to abort when it committed T_h, but it did so with a lower or equal label, so the flow created by the abort is safe. In the second case, it does not matter from an IFC perspective what the scheduler does: T_h and T_l have the same labels, so blocking or aborting one of them does not create any unsafe flows. Therefore, the scheduler should do whatever leads to the best performance in this case – most likely blocking T_l until T_h can commit.

In the third case, a new technique must be applied. Since T_h once had T_l's label pair, and since transaction conflict labels can never be lowered, T_l's label pair must be less restrictive than T_h's conflict label. Therefore, T_h can be aborted without violating property A.1. This fact provides the necessary intuition to write a rule for flow-safe scheduling for snapshot isolation:

Rule A.1. (Flow-Safe Scheduling for Snapshot Isolation)
If a transaction T_l attempts to write a tuple that was previously written by a concurrent transaction T_h, and if the processes issuing these transactions do not have the same labels, then T_h must be aborted so that T_l can proceed.

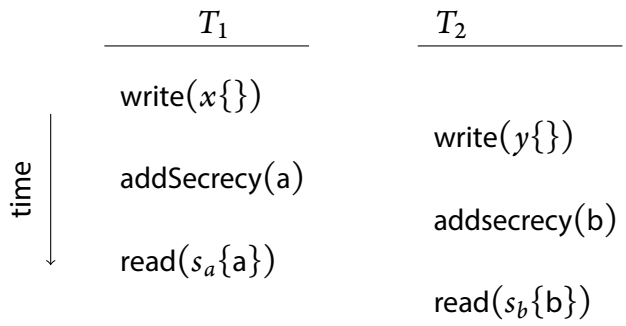
The rule handles the problem illustrated in figure A-1: the higher-labeled transaction T_h in the example will always be immediately aborted, so its behavior does not leak information to the lower-labeled transaction T_l. In a different example where the first process to write x lowers its label instead of raising it, rule A.1 also specifies that the earlier transaction be aborted. However, one could argue for aborting either transaction in that case.

Since a transaction that raises its label after writing a value x will always be aborted if a concurrent transaction writes x, starvation is possible. However, the problem only arises for processes that change their labels in the middle of a transaction in order to write tuples with different labels.

A.3 Serializable Flow-Safe Scheduling

Snapshot isolation is a weaker isolation model than serializability, so it would be desirable to support serializability as well. However, this appendix shows that property A.1 can't be guaranteed efficiently in a system with general transactions, the Aeolus model, and serializability.

Consider the following schedule of two transactions, T_1 and T_2. T_1 and T_2 both start with empty secrecy labels, and they write x and y, respectively. Then T_1 adds tag a to its secrecy label, while T_2 adds tag b. Next, T_1 reads a secret with label {a}, while T_2 reads a secret with label {b}.



Note that T_1 and T_2 don't have any reads or writes in common. Therefore, any reasonable scheduler would allow both transactions to run concurrently up to the point illustrated in the diagram. However, suppose that T_1 now tries to read y if the first bit of s_a is a 1 and T_2 tries to read x if the first bit of s_b is a 1. If both reads occur, then the resulting schedule isn't serializable: T_1 can't precede T_2 in the serial order because T_2 sees the old value of x, which doesn't include T_1's write, and T_2 can't come before T_1 for an analogous reason.

Furthermore, aborting T_1 leaks the first bit of s_b to the process that issued T_1, while aborting T_2 leaks the first bit of s_a to the process that issued T_2. A serializable scheduler would need to abort one transaction or the other, so it would be forced to accept the unsafe flow. Even some hypothetical scheduler that allowed dirty reads couldn't solve the problem. For instance, the scheduler might allow T_2 to see T_1's uncommitted write to x, which allows T_2 to be ordered after T_1. However, allowing dirty reads introduces a new problem: if T_1 aborts, then T_2 must abort as well, and this conveys information from T_1 to T_2.

Therefore, any solution to the unsafe flow problem must either limit concurrency (in particular, it must disallow the example illustrated above), or it must restrict

the transaction model. For instance, a policy that says that no two concurrent transactions may write any data with the same labels would work, but it severely limits concurrency. Similarly, a limited transaction model where the scheduler is allowed to analyze entire transactions in advance could lead to a solution, but this requires a different, more limited interface between applications and the database.

It was the observation of this problem that motivated the decision to use transaction clearance instead of flow-safe scheduling in section 6.2. Although the IFDB prototype is based on snapshot isolation and appendix A.2 shows that flow-safe scheduling is possible for snapshot isolation, semantics that fundamentally do not work for serializability are undesirable.

References

[1] Health insurance portability and accountability act (HIPAA) of 1996. US public law no. 104–191. HR 3103, 104th Congress.

[2] Air Force Studies Board, Committee on Multilevel Data Management Security. *Multilevel Data Management Security*. National Academy Press, Washington, DC, USA, March 1983.

[3] American National Standards Institute. *Information Technology – Database Languages – SQL*. Number ISO/IEC 9075-4:2011. New York, NY, USA, December 2011.

[4] Paul Ammann, Frank Jaeckle, and Sushil Jajodia. A two snapshot algorithm for concurrency control in multi-level secure databases. In *Proceedings of the 1992 IEEE Symposium on Security and Privacy (Oakland'92)*, pages 204–215, Oakland, CA, USA, May 1992. IEEE Computer Society.

[5] Paul Ammann, Sushil Jajodia, and Phyllis Frankl. Globally consistent event ordering in one-directional distributed environments. *IEEE Transactions on Parallel and Distributed Systems*, 7(6):665–670, June 1996.

[6] Aslan Askarov, Danfeng Zhang, and Andrew Myers. Predictive black-box mitigation of timing channels. In *17th ACM Conference on Computer and Communications Security (CCS'10)*, pages 297–307, Chicago, IL, USA, October 2010. ACM.

[7] Adam Barth, Anupam Datta, John Mitchell, and Helen Nissenbaum. Privacy and contextual integrity: Framework and applications. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (Oakland'06)*, pages 184–198, Oakland, CA, USA, May 2006. IEEE Computer Society.

[8] David Basin, Felix Klaedtke, and Samuel Müller. Monitoring security policies with metric first-order temporal logic. In *Proceedings of the 15th ACM Symposium on Access Control Models and Technologies (SACMAT'10)*, pages 23–34, Pittsburgh, PA, USA, June 2010. ACM.

[9] David Bell and Leonard LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, Electronic Systems Division, United States Air Force, Bedford, MA, USA, March 1976.

[10] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (SIGMOD'95)*, pages 1–10, San Jose, CA, USA, May 1995. ACM.

[11] Kenneth Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, Electronic Systems Division, United States Air Force, Bedford, MA, USA, April 1977.

[12] Barbara Blaustein, Sushil Jajodia, Catherine McCollum, and LouAnna Notargiacomo. A model of atomicity for multilevel transactions. In *Proceedings of the 1993 IEEE Symposium on Security and Privacy (Oakland'93)*, pages 120–134, Oakland, CA, USA, May 1993. IEEE Computer Society.

[13] Steven Bobrowski, Maria Pratt, and Timothy Smith. *Oracle7 Server Application Developer's Guide, Release 7.3*. Oracle Corporation, Redwood City, CA, USA, February 1996.

[14] Rebecca Bond, Kevin See, Carmen Wong, and Yuk-Kuen Chan. *Understanding DB2 9 Security*, chapter 6. IBM Press, Indianopolis, IN, USA, 1st edition, December 2006.

[15] Rae Burns and Yi-Fang Koh. A comparison of multilevel structured query language (SQL) implementations. In *Proceedings of the 12th Annual Computer*

Security Applications Conference, pages 192–202, San Diego, CA, USA, December 1996. IEEE Computer Society.

[16] Rui Chen, Noman Mohammed, Benjamin Fung, Bipin Desai, and Li Xiong. Publishing set-valued data via differential privacy. *Proceedings of the VLDB Endowment*, 4(11):1087–1098, August 2011.

[17] Winnie Cheng. *Information Flow for Secure Distributed Applications*. PhD, Massachusetts Institute of Technology, Cambridge, MA, August 2009. Also available as technical report MIT-CSAIL-TR-2009-040.

[18] Winnie Cheng, Dan Ports, David Schultz, Victoria Popic, Aaron Blankstein, James Cowling, Dorothy Curtis, Liuba Shrira, and Barbara Liskov. Abstractions for usable information flow control in Aeolus. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX'12)*, Boston, MA, USA, June 2012. USENIX Association.

[19] Adam Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*, pages 105–118, Vancouver, BC, Canada, October 2010. USENIX Association.

[20] Stephen Chong and Andrew Myers. Security policies for downgrading. In *11th ACM Conference on Computer and Communications Security (CCS'04)*, pages 198–209, Washington, DC, USA, October 2004. ACM.

[21] Stephen Chong, Krishnaprasad Vikram, and Andrew Myers. SIF: Enforcing confidentiality and integrity in web applications. In *Proceedings of the 16th USENIX Security Symposium (Security'07)*, pages 1–16, Boston, MA, USA, August 2007. USENIX Association.

[22] David Clark and David Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy (Oakland'87)*, pages 184–194, Oakland, CA, USA, April 1987. IEEE Computer Society.

[23] Privacy Rights Clearinghouse. Chronology of data breaches: Security breaches 2005–present. http://www.privacyrights.org/data-breach.

[24] Edgar Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[25] Edgar Codd. *The relational model for database management: Version 2*. Addison-Wesley Longman, Boston, MA, USA, 1990.

[26] Oliver Costich. Transaction processing using an untrusted scheduler in a multilevel database with replicated architecture. In *Database Security, V: Status and Prospects: Results of the IFIP WG 11.3 Workshop on Database Security (DBSEC 1991)*, pages 173–190, Shepherdstown, WV, USA, November 1991.

[27] Tore Dalenius and Steven Reiss. Data-swapping: A technique for disclosure control. *Journal of Statistical Planning and Inference*, 6(1):73–85, 1982.

[28] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: A flexible information flow architecture for software security. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA'07)*, pages 482–493, San Diego, CA, USA, June 2007. ACM.

[29] Christopher Date. *Relational Database: Selected Writings*. Addison-Wesley, 1986.

[30] Bernhard Debatin, Jennette Lovejoy, Ann-Kathrin Horn, and Brittany Hughes. Facebook and online privacy: Attitudes, behaviors, and unintended consequences. *Journal of Computer-Mediated Communication*, 15(1):83–108, November 2009.

[31] Dorothy Denning. *Secure Information Flow in Computer Systems*. PhD, Purdue University, West Lafayette, IN, May 1975.

[32] Dorothy Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[33] Dorothy Denning. Secure statistical databases with random sample queries. *ACM Transactions on Database Systems*, 5(3):291–315, September 1980.

[34] Dorothy Denning. *Cryptography and Data Security*, chapter 5: Information Flow Controls. Addison-Wesley, Boston, MA, USA, 1982.

[35] Dorothy Denning. Cryptographic checksums for multilevel database security. In *Proceedings of the 1984 IEEE Symposium on Security and Privacy (Oakland'84)*, pages 52–61, Oakland, CA, USA, April 1984. IEEE Computer Society.

[36] Dorothy Denning and Peter Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[37] Dorothy Denning and Teresa Lunt. The SeaView security model. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy (Oakland'88)*, pages 218–233, Oakland, CA, USA, April 1988. IEEE Computer Society.

[38] Dorothy Denning, Teresa Lunt, Roger Schell, Mark Heckman, and William Shockley. A multilevel relational data model. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy (Oakland'87)*, pages 220–234, Oakland, CA, USA, April 1987. IEEE Computer Society.

[39] *Department of Defense Trusted Computer System Evaluation Criteria*. Department of Defense, December 1985. DOD 5200.28-STD (the Orange Book).

[40] Blair Dillaway and J. Thomas Haigh. A practical design for a multilevel secure database management system. In *A Collection of Technical Papers: AIAA/ASIS/DODCI 2nd Aerospace Computer Security Conference*, pages 44–57, McLean, VA, USA, December 1986. AIAA.

[41] Deborah Downs and Gerald Popek. A kernel design for a secure data base management system. In *Proceedings of the 3rd International Conference on Very Large Data Bases (VLDB 1977)*, pages 507–514, Tokyo, Japan, October 1977. IEEE Computer Society and ACM.

[42] Cynthia Dwork. Differential privacy: A survey of results. In *Proceedings of the 5th International Conference on Theory and Applications of Models of Computation (TAMC 2008)*, pages 1–19, Xi'an, China, April 2008. Springer.

[43] Patricia Dwyer, Emmanuel Onuegbe, Paul Stachour, and Bhavani Thuraisingham. Query processing in LDV: A secure database system. In *Proceedings of the 4th Aerospace Computer Security Applications Conference*, pages 118–124, Orlando, FL, USA, December 1988. IEEE Computer Society.

[44] William Enck, Peter Gilbert, Byung-Gon Chun, Landon Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*, pages 393–408, Vancouver, BC, Canada, October 2010. USENIX Association.

[45] Federal Trade Commission. In the matter of Google, Inc. FTC Docket No. C-4336, October 2011.

[46] Steven Feuerstein and Bill Pribyl. *Oracle PL/SQL Programming*. Oracle Corporation, Redwood City, CA, USA, 3rd edition, September 2002.

[47] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 124, August 2004.

[48] Open Security Foundation. DataLossDB. http://datalossdb.org/.

[49] Judith Froscher and Catherine Meadows. Achieving a trusted database management system using parallelism. In *Database Security, II: Status and Prospects: Results of the IFIP WG 11.3 Workshop on Database Security (DBSEC 1988)*, pages 151–160, Kingston, ON, Canada, October 1988. Elsevier Science Publishers.

[50] Judith Froscher, Myong Kang, John McDermott, Oliver Costich, and Carl Landwehr. A practical approach to high assurance multilevel secure computing service. In *Proceedings of the 10th Annual Computer Security Applications Conference*, pages 2–11, Orlando, FL, USA, December 1994. IEEE Computer Society.

[51] Ariel Futoransky, Damián Saura, and Ariel Waissbein. The ND2DB attack: Database content extraction using timing attacks on the indexing algorithms. In *Proceedings of the First USENIX Workshop on Offensive Technologies (WOOT)*, Boston, MA, USA, August 2007. USENIX Association.

[52] George Gajnak. Some results from the entity/relationship multilevel secure DBMS project. In *Proceedings of the 4th Aerospace Computer Security Applications Conference*, pages 66–71, Orlando, FL, USA, December 1988. IEEE Computer Society.

[53] Moses Garuba. Performance study of a COTS distributed DBMS adapted for multilevel security. Technical Report RHUL-MA-2004-2, Royal Holloway, University of London, Egham, Surrey, UK, July 2004.

[54] Cristi Garvey and Philip Papaccio. Multilevel data store design. In *A Collection of Technical Papers: AIAA/ASIS/DODCI 2nd Aerospace Computer Security Conference*, pages 58–64, McLean, VA, USA, December 1986. AIAA.

[55] Cristi Garvey and Amy Wu. ASD_Views. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy (Oakland'88)*, pages 85–95, Oakland, CA, USA, April 1988. IEEE Computer Society.

[56] PostgreSQL Global Development Group. *PostgreSQL 9.1 Documentation*, September 2011. http://www.postgresql.org/docs/9.1/static/.

[57] Joseph Goguen and José Meseguer. Security policies and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy (Oakland'82)*, pages 11–20, Oakland, CA, USA, April 1982. IEEE Computer Society.

[58] Richard Graubart. The integrity-lock approach to secure database management. In *Proceedings of the 1984 IEEE Symposium on Security and Privacy (Oakland'84)*, pages 62–74, Oakland, CA, USA, April 1984. IEEE Computer Society.

[59] Richard Graubart and Kevin Duffy. Design overview for retrofitting integrity-lock architecture onto a commercial DBMS. In *Proceedings of the 1985 IEEE Symposium on Security and Privacy (Oakland'85)*, pages 147–159, Oakland, CA, USA, April 1985. IEEE Computer Society.

[60] Richard Graubart and John Woodward. A preliminary naval surveillance DBMS security model. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy (Oakland'82)*, pages 21–37, Oakland, CA, USA, April 1982. IEEE Computer Society.

[61] Jim Gray, Raymond Lorie, Gianfranco Putzolu, and Irving Traiger. Granularity of locks and degrees of consistency in a shared data base. In *Proceedings of the IFIP Working Conference on Modelling in Data Base Management Systems*, pages 365–394, Freudenstadt, Germany, January 1976. North-Holland.

[62] Michael Grohn. A model of a protected data management system. Technical Report ESD-TR-76-289, Electronic Systems Division, United States Air Force, Bedford, MA, USA, June 1976.

[63] Hakan Hacıgümüş, Bala Iyer, Chen Li, and Sharad Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD'02)*, pages 216–227, Madison, WI, USA, June 2002. ACM.

[64] William Halfond and Alessandro Orso. AMNESIA: Analysis and monitoring for neutralizing SQL-injection attacks. In *Proceedings of the IEEE and ACM International Conference on Automated Software Engineering (ASE 2005)*, pages 174–183, Long Beach, CA, November 2005.

[65] Michael Hay, Vibhor Rastogi, Gerome Miklau, and Dan Suciu. Boosting the accuracy of differentially private histograms through consistency. *Proceedings of the VLDB Endowment*, 3(1–2):1021–1032, September 2010.

[66] Joseph Hellerstein and Avi Pfeffer. The RD-Tree: An index structure for sets. Technical Report 1252, University of Wisconsin, Madison, WI, USA, November 1994.

[67] Thomas Hinke. Secure database management system: Architectural analysis. In *A Collection of Technical Papers: AIAA/ASIS/DODCI 2nd Aerospace Computer Security Conference*, pages 65–72, McLean, VA, USA, December 1986. AIAA.

[68] Thomas Hinke and Marvin Schaefer. Secure data management system. Technical Report RADC-TR-75-266, Rome Air Development Center, United States Air Force, Rome, NY, USA, November 1975.

[69] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th International World Wide Web Conference (WWW'04)*, pages 40–52, New York, NY, May 2004. ACM.

[70] Bret Hull, Vladimir Bychkovsky, Yang Zhang, Kevin Chen, Michel Goraczko, Allen Miu, Eugene Shih, Hari Balakrishnan, and Samuel Madden. CarTel: A distributed mobile sensor computing system. In *Proceedings of the 4th*

International Conference on Embedded Network Sensor Systems (SenSys'06), pages 125–138, Boulder, CO, USA, November 2006. ACM.

[71] Galen Hunt and James Larus. Singularity: Rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, April 2007.

[72] Informix Software Inc. *Informix-OnLine/Secure Security Features Guide*. Menlo Park, CA, USA, April 1993.

[73] Infosystems Technology, Inc. *Trusted RUBIX Version 6 Trusted Facility Manual*. Gaithersburg, MD, USA, 7th edition, 2011.

[74] Sushil Jajodia and Vajayalakshmi Atluri. Alternative correctness criteria for concurrent execution of transactions in multilevel secure databases. In *Proceedings of the 1992 IEEE Symposium on Security and Privacy (Oakland'92)*, pages 216–224, Oakland, CA, USA, May 1992. IEEE Computer Society.

[75] Sushil Jajodia and Ravi Sandhu. Polyinstantiation integrity in multilevel relations. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy (Oakland'90)*, pages 104–115, Oakland, CA, USA, May 1990. IEEE Computer Society.

[76] Sumit Jeloka. *Oracle Label Security Administrator's Guide, 11g Release 2*. Oracle Corporation, 2009.

[77] Iwen Kang and Thomas Keefe. On transaction processing for multilevel secure replicated databases. In *Proceedings of the 2nd European Symposium On Research In Computer Security (ESORICS 1992)*, volume 648 of *Lecture Notes in Computer Science*, pages 329–347. Springer, Tolouse, France, November 1992.

[78] Paul Karger and John Wray. Storage channels in disk arm optimization. In *Proceedings of the 1991 IEEE Symposium on Security and Privacy (Oakland'91)*, pages 52–61, Oakland, CA, USA, May 1991. IEEE Computer Society.

[79] Thomas Keefe and Wei-Tek Tsai. Multiversion concurrency control for multilevel secure database systems. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy (Oakland'90)*, pages 369–383, Oakland, CA, USA, May 1990. IEEE Computer Society.

[80] Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. Implicit flows: Can't live with 'em, can't live without 'em. In *Proceedings of the International Conference on Information Systems Security (ICISS)*, volume 5352 of *Lecture Notes in Computer Science*, pages 56–70, Hyderabad, India, December 2008. Springer.

[81] Gillian Kirkby and Michael Grohn. The reference monitor technique for security in data base management systems. *IEEE Data Base Engineering Bulletin*, 1(2):8–16, June 1977.

[82] KaiGai Kohei. SE-PostgreSQL online documentation. http://wiki.postgresql.org/wiki/SEPostgreSQL.

[83] Eddie Kohler. Hot crap! In *Proceedings of the Workshop on Organizing Workshops, Conferences, and Symposia for Computer Systems (WOWCS'08)*, San Francisco, CA, USA, April 2008. USENIX Association.

[84] Maxwell Krohn. *Information Flow Control for Secure Web Sites*. PhD, Massachusetts Institute of Technology, Cambridge, MA, USA, September 2008.

[85] Maxwell Krohn, Petros Efstathopoulos, Cliff Frey, Frans Kaashoek, Eddie Kohler, David Mazières, Robert Morris, Michelle Osborne, Steve VanDeBogart, and David Ziegler. Make least privilege a right (not a privilege). In *10th Workshop on Hot Topics in Operating Systems (HotOS X)*, Santa Fe, NM, USA, June 2005. USENIX Association.

[86] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, pages 321–334, Stevenson, WA, USA, October 2007. ACM.

[87] Hsiang-Tsung Kung and John Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[88] Monica Lam, Michael Martin, Benjamin Livshits, and John Whaley. Securing web applications with static and dynamic information flow tracking. In

Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '08), pages 3–12, San Francisco, CA, USA, January 2008. ACM.

[89] Butler Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[90] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, November 1992.

[91] Sang-Won Lee, Yong-Han Kim, and Hyoung-Joo Kim. The semantics of an extended referential integrity for a multilevel secure relational data model. *Data and Knowledge Engineering*, 48(1):129–152, 2004.

[92] Kristen LeFevre, Rakesh Agrawal, Vuk Ercegovac, Raghu Ramakrishnan, Yirong Xu, and David DeWitt. Limiting disclosure in hippocratic databases. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB 2004)*, pages 108–119, Toronto, Canada, September 2004. Morgan Kaufmann.

[93] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*, pages 158–170, Long Beach, CA, USA, January 2005. ACM.

[94] Peng Li and Steve Zdancewic. Practical information-flow control in web-based information systems. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop*, pages 2–15, Aix-en-Provence, France, June 2005. IEEE Computer Society.

[95] Jed Liu and Andrew Myers. Defining and enforcing referential security. In submission, April 2012.

[96] Jed Liu, Michael George, Krishnaprasad Vikram, Xin Qi, Lucas Waye, and Andrew Myers. Fabric: A platform for secure distributed computation and storage. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09)*, pages 321–334, Big Sky, MT, USA, October 2009. ACM.

[97] Yali Liu, Dipak Ghosal, Frederik Armknecht, Ahmad-Reza Sadeghi, Steffen Schulz, and Stefan Katzenbeisser. Hide and seek in time: Robust covert timing channels. In *Proceedings of the 14th European Symposium On Research In Computer Security (ESORICS 2009)*, volume 5789 of *Lecture Notes in Computer Science*, pages 120–135, Saint-Malo, France, September 2009. Springer.

[98] V. Benjamin Livshits and Monica Lam. Finding security vulnerabilities in Java applications with static analysis. In *Proceedings of the 14th USENIX Security Symposium (Security'05)*, Baltimore, MD, USA, August 2005. USENIX Association.

[99] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 29–42, Boston, MA, USA, June 2001. USENIX Association.

[100] Teresa Lunt. Security in database systems: A research perspective. *Computers and Security*, 11(1):41–56, March 1992.

[101] Teresa Lunt, Dorothy Denning, Roger Schell, Mark Heckman, and William Shockley. The SeaView security model. *IEEE Transactions on Software Engineering*, 16(6):593–607, June 1990.

[102] Ashwin Machanavajjhala, Daniel Kifer, Johannes Gehrke, and Muthuramakrishnan Venkitasubramaniam. ℓ-diversity: Privacy beyond k-anonymity. *ACM Transactions on Knowledge Discovery from Data*, 1(1), March 2007.

[103] William Maimone and Ira Greenberg. Single-level multiversion schedulers for multilevel secure database systems. In *Proceedings of the 6th Annual Computer Security Applications Conference*, pages 137–147, Tucson, AZ, USA, December 1990. IEEE Computer Society.

[104] Amit Mathur and Thomas Keefe. The concurrency control and recovery problem for multilevel update transactions in MLS systems. In *Proceedings of the Workshop on Computer Security Foundations VI*, pages 10–23, June 1993.

[105] John McDermott, Sushil Jajodia, and Ravi Sandhu. A single-level scheduler for the replicated architecture for multilevel-secure databases. In *Proceedings*

of the 7th Annual Computer Security Applications Conference, pages 2–11, San Antonio, TX, USA, December 1991. IEEE Computer Society.

[106] Douglas McIlroy and James Reeds. Multilevel security in the UNIX tradition. *Software: Practice and Experience*, 22(8), August 1992.

[107] Frank McSherry. Privacy integrated queries: An extensible platform for privacy-preserving data analysis. In *Proceedings of the 35th ACM SIGMOD International Conference on Management of Data (SIGMOD'09)*, pages 19–30, Providence, RI, USA, June 2009. ACM.

[108] Jonathan Millen. 20 years of covert channel modeling and analysis. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy (Oakland'99)*, pages 113–114, Oakland, CA, USA, May 1999. IEEE Computer Society.

[109] Jonathan Millen and Teresa Lunt. Security for object-oriented database systems. In *Proceedings of the 1992 IEEE Symposium on Security and Privacy (Oakland'92)*, pages 260–272, Oakland, CA, USA, May 1992. IEEE Computer Society.

[110] MITRE Corporation. Information exposure through an error message. CWE-209, May 2012. http://cwe.mitre.org/data/definitions/209.html.

[111] Tom Murphy. Security glitch exposes WellPoint customers' financial, medical data. *USA Today*, June 29, 2010. http://www.usatoday.com/money/industries/health/2010-06-29-wellpoint-data-breach_N.htm.

[112] Andrew Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'06)*, pages 228–241, San Antonio, TX, USA, January 1999. ACM.

[113] Andrew Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP'97)*, pages 129–142, Saint Malo, France, October 1997. ACM.

[114] National Institute of Standards and Technology. National vulnerability database. http://nvd.nist.gov/.

[115] Patrick Pacious. Re: Incident notification. Choice Hotels International compulsory data breach notification to the State of California's Office of the Attorney General. https://oag.ca.gov/system/files/Choice%20Individual%20Notification%20(Master)_0.pdf, April 2012.

[116] Shankar Pal. A locking protocol for multilevel secure databases using two committed versions. In *Proceedings of the 10th Annual Conference on Computer Assurance (COMPASS '95)*, pages 197–210, Gaithersburg, MD, USA, June 1995.

[117] Chanjung Park, Seog Park, and Sang Son. Multiversion locking protocol with freezing for secure real-time database systems. *IEEE Transactions on Knowledge and Data Engineering*, 14(5):1141–1154, October 2002.

[118] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID'05)*, pages 124–145, Seattle, WA, USA, September 2005. Springer.

[119] Raluca Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*, pages 85–100, Cascais, Portugal, October 2011. ACM.

[120] Chandramouli Ramaswamy and Ravi Sandhu. Role based access control features in commercial database management systems. In *21st National Information Systems Security Conference*, Arlington, VA, USA, October 1998. National Institute of Standards and Technology.

[121] Indrajit Ray, Elisa Bertino, Sushil Jajodia, and Luigi Mancini. An advanced commit protocol for MLS distributed database systems. In *3rd ACM Conference on Computer and Communications Security (CCS'96)*, pages 119–128, New Delhi, India, March 1996. ACM.

[122] David Reed. *Naming and Synchronization in a Decentralized Computer System*. PhD, Massachusetts Institute of Technology, Cambridge, MA, USA, September 1978. Available from http://dspace.mit.edu/handle/1721.1/16279.

[123] Shariq Rizvi, Alberto Mendelzon, Sundararajarao Sudarshan, and Prasan Roy. Extending query rewriting techniques for fine-grained access control. In

Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD'04), pages 551–562, Paris, France, June 2004. ACM.

[124] Pankaj Rohatgi. Side-channel attacks. In *Handbook of Information Security*, volume 3, pages 241–260. Wiley, 2005.

[125] Indrajit Roy, Donald Porter, Michael Bond, Kathryn McKinley, and Emmett Witchel. Laminar: Practical fine-grained decentralized information flow control. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*, pages 63–74, Dublin, Ireland, June 2009. ACM.

[126] Jerome Saltzer. Protection and the control of information sharing in Multics. In *Proceedings of the 4th ACM Symposium on Operating Systems Principles (SOSP'73)*, pages 10–24, Yorktown Heights, NY, USA, October 1973. ACM.

[127] Jerome Saltzer and Michael Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE*, volume 63, pages 1278–1308, Piscataway, NJ, USA, September 1975. IEEE.

[128] Ravi Sandhu and Fang Chen. The multilevel relational (MLR) data model. *ACM Transactions on Information and System Security*, 1(1):93–132, November 1998.

[129] Ravi Sandhu and Sushil Jajodia. Referential integrity in multilevel secure databases. In *Proceedings of the 16th NIST-NCSC National Computer Security Conference*, pages 39–52, Baltimore, MD, USA, September 1993.

[130] Nelson Schwartz and Eric Dash. Thieves found Citigroup site an easy entry. *The New York Times*, June 13, 2011. URL https://www.nytimes.com/2011/06/14/technology/14security.html.

[131] Kenneth Smith and Marianne Winslett. Entity modeling in the MLS relational model. In *Proceedings of the 18th International Conference on Very Large Data Bases (VLDB 1992)*, pages 199–210, Vancouver, Canada, August 1992. Morgan Kaufmann.

[132] Kenneth Smith, Barbara Blaustein, Sushil Jajodia, and LouAnna Notargiacomo. Correctness criteria for multilevel secure transactions. *IEEE Transactions on Knowledge and Data Engineering*, 8:32–45, 1996.

[133] Paul Stachour and Bhavani Thuraisingham. Design of LDV: A multilevel secure relational database management system. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):190–209, June 1990.

[134] Zachary Stengel and Tevfik Bultan. Analyzing Singularity channel contracts. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA '09)*, pages 13–24, Chicago, IL, USA, July 2009. ACM.

[135] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'06)*, pages 372–382, Charleston, SC, USA, January 2006. ACM.

[136] G. Edward Suh, Jae Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 85–96, Boston, MA, USA, October 2004. ACM.

[137] Latanya Sweeney. *k*-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness, and Knowledge-Based Systems*, 10(5):557–570, October 2002.

[138] Sybase, Inc. *Final Evaluation Report: SQL Server Version 11.0.6 and Secure SQL Server Version 11.0.6*, chapter 5: Security Architecture. National Computer Security Center, Ft. Meade, MD, USA, March 1997.

[139] Transaction Processing Performance Council. *TPC Benchmark W (Web Commerce) Specification, revision 1.8*. San Jose, CA, USA, February 2002.

[140] Transaction Processing Performance Council. *TPC Benchmark C, revision 5.9*. San Jose, CA, USA, June 2007.

[141] Steve Vandebogart, Petros Efstathopoulos, Eddie Kohler, Maxwell Krohn, Cliff Frey, David Ziegler, Frans Kaashoek, Robert Morris, and David Mazières. Labels and event processes in the Asbestos operating system. *ACM Transactions on Computer Systems*, 25(4), December 2007.

[142] Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Sam Madden. Tolerating Byzantine faults in database systems using commit barrier scheduling. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, pages 59–72, Stevenson, WA, USA, October 2007. ACM.

[143] Linda Vetter, Gordon Smith, and Teresa Lunt. TCB subsets: The next step. In *Proceedings of the 5th Annual Computer Security Applications Conference*, pages 216–221, Tucson, AZ, USA, December 1989. IEEE Computer Society.

[144] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2–3):167–187, January 1996.

[145] Dan Wallach, Dirk Balfanz, Drew Dean, and Edward Felten. Extensible security architectures for Java. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP'97)*, pages 116–128, Saint Malo, France, October 1997. ACM.

[146] Clark Weissman. Security controls in the ADEPT-50 time-sharing system. In *Proceedings of the Fall Joint Computer Conference (AFIPS '69)*, pages 119–133, Las Vegas, NV, USA, November 1969. AFIPS Press.

[147] Suzanne Widup. *The Leaking Vault: Five Years of Data Breaches*. Digital Forensics Association, July 2010. http://www.digitalforensicsassociation.org/storage/The_Leaking_Vault-Five_Years_of_Data_Breaches.pdf.

[148] Simon Wiseman. Using SWORD for the military airlift command example database. In *Database Security, VI: Status and Prospects: Results of the IFIP WG 11.3 Workshop on Database Security (DBSEC 1992)*, pages 73–88, Vancouver, BC, Canada, August 1992. Elsevier Science Publishers.

[149] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*, pages 85–96, Philadelphia, PA, USA, January 2012. ACM.

[150] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Improving application security with data flow assertions. In *Proceedings of the*

22nd ACM Symposium on Operating Systems Principles (SOSP'09), pages 291–304, Big Sky, MT, USA, October 2009. ACM.

[151] Sachiko Yoshihama, Takeo Yoshizawa, Yuji Watanabe, Michiharu Kudoh, and Kazuko Oyanagi. Dynamic information flow control architecture for web applications. In *Proceedings of the 12th European Symposium On Research In Computer Security (ESORICS 2007)*, volume 4734 of *Lecture Notes in Computer Science*, pages 267–282, Dresden, Germany, September 2007. Springer.

[152] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*, pages 263–278, Seattle, WA, USA, November 2006. USENIX Association.

[153] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*, pages 293–308, San Francisco, CA, USA, April 2008. USENIX Association.